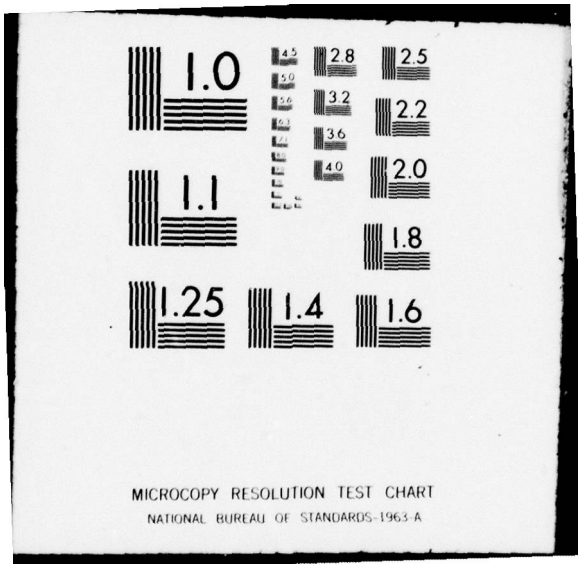
AD-A060 495 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
PERFORMANCE EVALUATION OF MULTIPLE PROCESSOR SYSTEMS.(U)
AUG 78 L RASKIN
UNCLASSIFIED CMU-CS-78-141 F44620-73-C-0074
NL

1 OF 3
AD
A060495


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LEVEL II

CMU-CS-78-141

Performance Evaluation of Multiple Processor Systems

Levy Raskin

August 1978

DEPARTMENT of COMPUTER SCIENCE

Carnegie-Mellon University

(14) CMU-CS-78-141

(9) Interim rept.

(6) # Performance Evaluation of Multiple Processor Systems.

(10) Levy/Raskin

Department of Electrical Engineering,
Carnegie-Mellon University,
Pittsburgh, Pennsylvania 15213.

(11) Aug[illegible] 1978

(12) 236p.

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was mainly supported by a research grant from the Government of Israel - ADA. The Cm* - Distributed Processing project was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract (15) F44620-73-C-0074, N00014-77-C-0500 which is monitored by the Air Force Office of Scientific Research, in part by the National Science Foundation Grant GJ 32758X, and in part by the Office of Naval Research contract N00014-77-C-0500. The LSI-11's and related equipment were supplied by the Digital Equipment Corporation.

403081

Performance Evaluation of Multiple Processor Systems

ABSTRACT

Technological trends in semiconductor and micro-processor development are clearly leading towards the production of a "computer on a chip". In the near future such a *computer module* will include the equivalent of today's mini-computer with some memory. Connecting many computer-modules together will probably be a cost-effective way to build high performance computer systems. This thesis investigates and contrasts the performance of two multiple computer-module structures: a multiprocessor with shared memory and a local computer network in which all communication is via messages. Results are derived both from running benchmark applications and from performance models.

Very little knowledge and experience has been gained on the performance of actual multiple processor applications. To investigate the problems and potential of these computer-module structures, an multiple processor system -- Cm* -- has been built at Carnegie-Mellon University. Firmware changes enable the emulation of both an efficient multiprocessor and a local computer network. Experiments were conducted on both types of structures to obtain performance information. Our practical methodology includes measurement of performance parameters using a set of benchmark application programs executing on Cm*, and performance models that were derived and validated - using the measurement results - and then applied for the performance investigation.

Measurements of the multiprocessor structure formed the basis of a detailed performance investigation of the initial, ten processor, Cm* system. This investigation showed the possibility of efficiently solving several types of problems using a multiprocessor. The various performance measures obtained - in particular, the relative access frequencies to various memory patterns e.g. code, global data etc. - gave insight into the important characteristics that make an application suitable for the multiprocessor structure, and into algorithm decomposition issues.

Several multiprocessor performance models were developed; their merits and deficiencies were explored. The main model is a queuing network model, with several classes of customers. Using classes of customers has allowed the more accurate modelling of many of the details and sequences of actions on the Cm* buses. Validation of the queuing network model against the measurements of one to eight computer-modules Cm*, showed that it accurately predicts the observed performance, and is efficient and straightforward to use. The model was then used to extrapolate performance for systems with 50 processors, faster processors, and faster buses - to explore the viability of the Cm* structure in much larger and more powerful configurations. Some theoretical aspects and proofs of efficient

computational algorithms, that were used in the model development, are also presented.

This work is a pioneer attempt to use the potential concurrency of a local computer network to achieve performance gains for applications that require close cooperation between several concurrent processes. The potential performance gain, by utilizing several processors, is already known for multiprocessors - but to date has not been seriously investigated for the local network structure. The detailed measurements provide the main performance parameters that were used to construct an analytic queuing network performance model. The model was shown to represent, quite accurately, the performance of an application running on a local network structure. The relative performance, and performance issues, involved in comparing the multiprocessor performance to the local network performance, are discussed and the advantages of using the shared address space in multiprocessors is shown. The design space attributes that mainly affect performance of local network structures are described. The possibility of forming an optimal match between application programs and multiple processor structures is examined.

Acknowledgements

Foremost it is a pleasure to thank my advisor, Sam Fuller. His many suggestions, encouragements, and support all along helped to shape the thesis and enabled its completion. I am grateful to the members of my thesis committee - Lloyd Dickman, Anita Jones, Alice Parker, and Dan Siewiorek - for helpful discussions and criticisms that improved the quality of the work.

The Computer Science Department at CMU, in particular the members of the Cm* group in which I was involved, created an excellent research atmosphere. I am especially indebted to Ivor Durham, Peter Feiler, John Ousterhout and Richard Swan, from this group, for useful discussions and help.

I would also like to thank Egon Balas, Gerard Baudet, Peter Oleinick and Bob Sproull for illuminating discussions during various phases of the research. The research was made possible by support from the Government of Israel - ADA. The Cm* project was supported by ARPA, NSF, ONR, and DEC. Their support is gratefully acknowledged.

Special thanks are due to my wife, Henia, for her continual encouragements and understanding during these long, and sometimes difficult, three years away from home.

Table of Contents

1. Introduction

1.1 Goals and Definitions

The goal of this thesis is to analyze and evaluate key performance issues in the use of multiple processor-memory pairs in the multiprocessor structure/local computer network structure space. A practical methodology and several techniques and tools were developed and applied to the performance investigation of these structures. These tools facilitate the investigation of each alternative structure by itself. However our main interest was to compare and evaluate the performance improvement that results from main memory sharing (in multiprocessors) with the network message communication protocol. The research vehicles included experiments and measurements of benchmark programs on the two multiple processor structures, and models that were developed, validated and applied in the performance investigation.

Technological trends in the semiconductor (LSI, VLSI) and micro-processor development, in the last few years, are clearly leading toward the production of a "Computer on a Chip". In the near future such a module will include the equivalent of today's minicomputer and some memory on a single LSI chip. Such a structure will be termed a Computer-Module. To investigate the problems and potential of computer module structures, an multiple Micro-Computer-Module structure -- Cm* -- has been built at Carnegie-Mellon university [Fuller 77a]. Firmware changes enable the emulation of both an efficient multiprocessor and a computer network.

Multiple processor systems, those that belong to the Multiple Instruction, Multiple Data stream (MIMD) category [Flynn 72], span a wide spectrum of structures with varying degrees of decentralization. This spectrum ranges all the way from loosely coupled and geographically distributed computer networks (e.g. the Arpanet), through tightly coupled (local) computer network (e.g. the Ethernet), and finally multiprocessor and multiple arithmetic unit processor computer systems. Figure 1.1 illustrates the range of the degree of coupling for these structures (which has been defined in [Fuller 78] as the worst case access time to shared data). Cm* configured as a multiprocessor is shown as the shaded area.

Our interest is in the differential in performance between multiprocessors and local computer networks. Two main characteristics that distinguish multiprocessors from computer networks are the mechanism to share data (and control) structures and the communication rate (bits/sec) between processors. Computer networks are characterized by interprocess message transfers as the data and control communication mechanism, while in multiprocessors communication is achieved by sharing the primary memory between all processors.

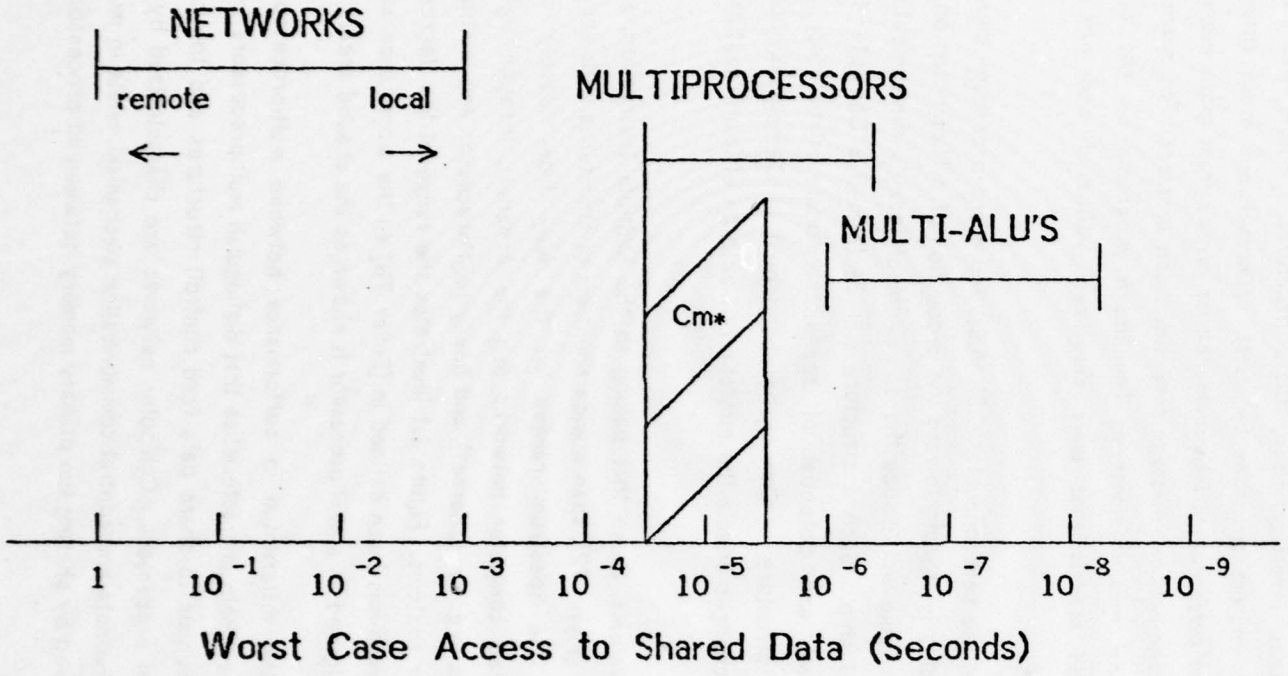


Fig. 1.1: Multiple Processor Systems - Range of Degree of Coupling

range.spx/b

Communication rate is the other main factor. Informal definition puts the local computer network as that part of the distributed computer spectrum between 0.1 MBPS (communication rate in Mega Bits Per Second) to 10 MBPS[1] - While remote computer networks and multiprocessors are on either side of local networks in the spectrum.

1.2 Methodology

To accomplish the goals of the thesis the performance of both multiple computer-module multiprocessors and local networks was investigated using measurements of benchmark programs on actual systems and using performance models.

Due to the almost total lack of information and experience on such structures a practical methodology had to be devised. The methodology that was used involved the following:

- A set of scientific oriented benchmark programs was written as the workload model to exercise the different structures. These programs were from different application areas and were used to stress the different aspects of the architecture on which they were executing.

- Detailed measurements were made and various performance measures (i.e. performance parameters upon which the performance evaluation was based) were collected. The Cm* system connected as a multiprocessor system was used for this investigation.

- Practical methods and tools were developed to enable exploring the performance of the multiprocessor system in greater depth (while changing system parameters) than could be done in the actual Cm* system. Analytic and simulation performance models were used. This involved proofs of some theoretical models and of efficient computational algorithms using a somewhat different viewpoint than, and extension to, the work of others. The derivation of these models included a trade-off between simplicity of the models and accuracy of the results.

- The performance models were validated using the results of the measurements on Cm*. For some of the models such a detailed validation had never been done before, and thus advantages and limitations of the models were discovered. An example of applying this models to the investigation of different extensions of the Cm* multiprocessor computer system are presented.

- Constructing and validating the multiprocessor computer models and exploring the performance issues enabled us to compare the multiprocessor structure to the local computer network structure. There is surprisingly little knowledge, or even research activity, regarding performance issues for the latter structure, i.e. using the local network as a vehicle for gainning performance by implementing cooperating parallel processes communicating by messages. The

[1] Metcalfe [Metcalfe 76] has observed that the communication rate/distance between processors product is a good measure of the state of the art and that the current state of the art is about 1.0 Gigameter-bits/second.

Cm* system was used as an emulation of a local computer network, using microcode in the central control processor in the structure to emulate the hardware network mechanisms (i.e. serial line + bus interface unit) and software support routine to handle the message communication protocols. Two benchmark application programs, that were previousiy measured on the Cm* multiprocessor structure, were adapted to the network environment and their performance was measured and compared to the multiprocessor performance. Several other issues concerning performance on a network structure were investigated.

1.3 Research Vehicles

1.3.1 CM* - The Computer-module Computer system

1.3.1.1 Cm* as a multiprocessor system

The distinguishing characteristic of multiprocessors is that all the processors in the structure share primary memory. The concept of multiprocessors is not new; the Burroughs D825 (1962), Bendix G-21 (1963), GE645 (1969), and IBM 360/65 (1969) provide early examples. In these multiprocessors conventional, relatively expensive, central processors were used, making it uneconomical to have more than a few processors. More recently, multiprocessors using minicomputers have been implemented [Heart 73],[Wulf 72], with as many as 14 - 16 processors in a single computer system. General surveys of multiprocessor's taxonomies and structures are given in [Flynn 72], [Baer 73], [Enslow 77a], [Jensen 77], [Fuller 78], [Swan 78].

The structure of the Cm* multiprocessor system grew from economic, performance, reliability and modularity considerations. Multiple processor structures with hundreds, and even thousands, processing elements were envisioned. A detailed description of Cm* and the design issues involved is given in [Fuller 77a], [Fuller 77b], [Fuller 78], [Jones 78], and a short description is repeated here.

The structure of Cm* is depicted in Fig. 1.2. The fundamental unit of Cm* is a computer module (Cm). Each Cm consists of a processing element (an DEC LSI-11 microprocessor), local memory, input/output devices, and a local switch (Slocal) which provides a simple interface between the Cm and the rest of the system. The primary memory of the system consists exclusively of the local memory of the Cm's.

A processor may directly reference any location in main memory. The Slocal uses simple mapping tables to decide on a reference-by-reference basis whether the physical address being referenced to is in the local memory. If it is, the Slocal perform a simple mapping function and the reference proceed very quickly. If it is not, the Slocal passes the reference

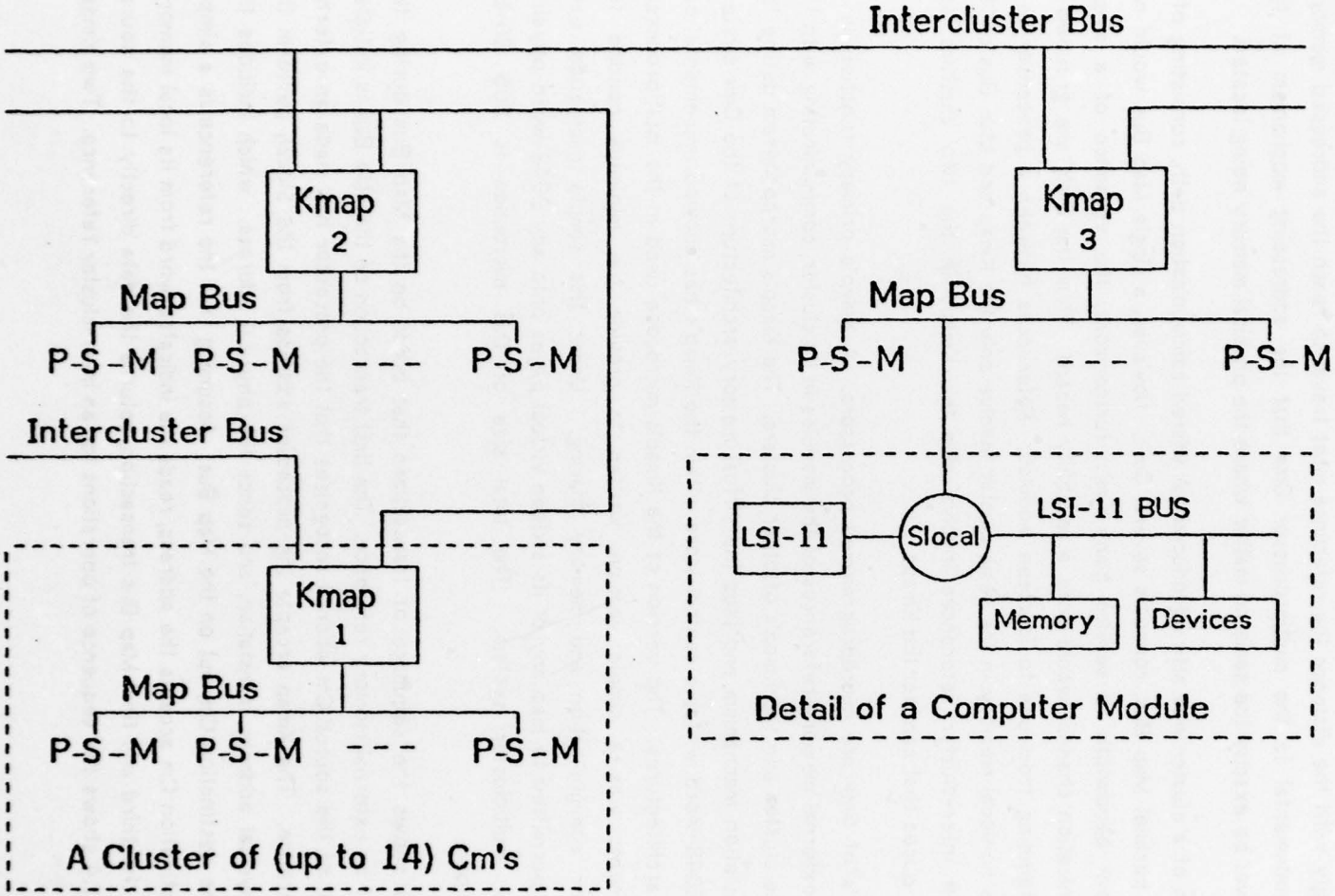


Fig. 1.2: Cm* - A Multiple Processor Structure

t2b.spx/b

to a mapping controller (Kmap). The Kmap's, which comprise a distributed processor/memory switch, communicate with each other and the Slocal's of the system to perform non local references for processors. The fact that a memory reference is non-local is completely transparent to the processor. While the reference is being performed by the Kmap and Slocal's, the processor waits just as if the reference was local. The duration of this wait varies strongly with the *distance* the reference must travel to reach the addressed memory, but is is fundamental to the multiprocessor Cm* that the addressing mechanism at the processor level be exactly the same no matter where the physical memory being located.

The notion of a *cluster* was also introduced. A shared communication path, consisting of a Kmap and a parallel Map Bus, connects several Cm's. However, a single Map Bus would not have sufficient bandwidth to service many Cm's; furthermore, the presence of a single inter-communication channel would pose a reliability hazard. Thus the Cm's are grouped in clusters containning from one to fourteen modules. References between a processor in a cluster and a nonlocal memory in the same cluster involve only the Kmap and Map Bus of the cluster, while inter-cluster references involve also the Kmaps in the two clusters and inter-cluster buses that connect the Kmaps.

The Kmap's of Cm* are microprogrammed processors. A Kmap's primary function is to process the external memory references of the modules in its cluster, communicating with the Slocal's of the cluster and the Kmap's of other clusters. The Kmap's microprogram define the address translation mechanism and thus the virtual memory architecture of the Cm* system. The use of 2048-word writable control store within the Kmap's has allowed implementations of different architectures. The version of the Kmap's microcode used in the multiprocessor Cm* experiments was a simple version, written to provide the minimum needed for interprocessor communication and memory sharing. Under this simple microcode each processor is permitted to map any of its sixteen virtual pages onto any 2048-word physical page in the multicluster system. The total size of this microcode is 505 80-bit microinstructions.

Figure 1.3 shows the sequence of transactions that occur on the Map Bus during the processing of an external memory reference. The first transaction on the Map Bus is initiated by the Slocal of the source Cm when it recognizes that the processor has made an external memory reference. The Kmap accepts the processor address from the Slocal, perform the virtual to physical address translation, and sends the physical address. which includes the number of the destination Cm, out on the Map Bus. Assuming that the reference is a simple read, the destination Cm accepts the address, reads the indicated word from its local memory, and then, in the third and final Map Bus transaction, returns the data directly to the source Cm. Figure 1.4 shows the sequence of operations for an inter-cluster reference. Two Kmaps

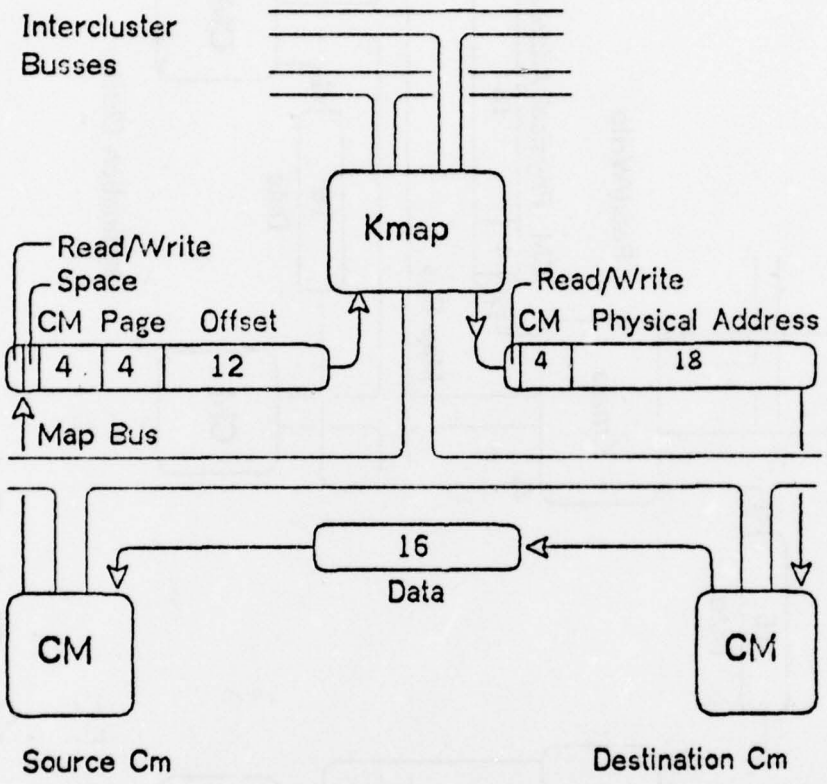


Fig. 1.3: Example of an Intra-Cluster Reference

12e.spx/b

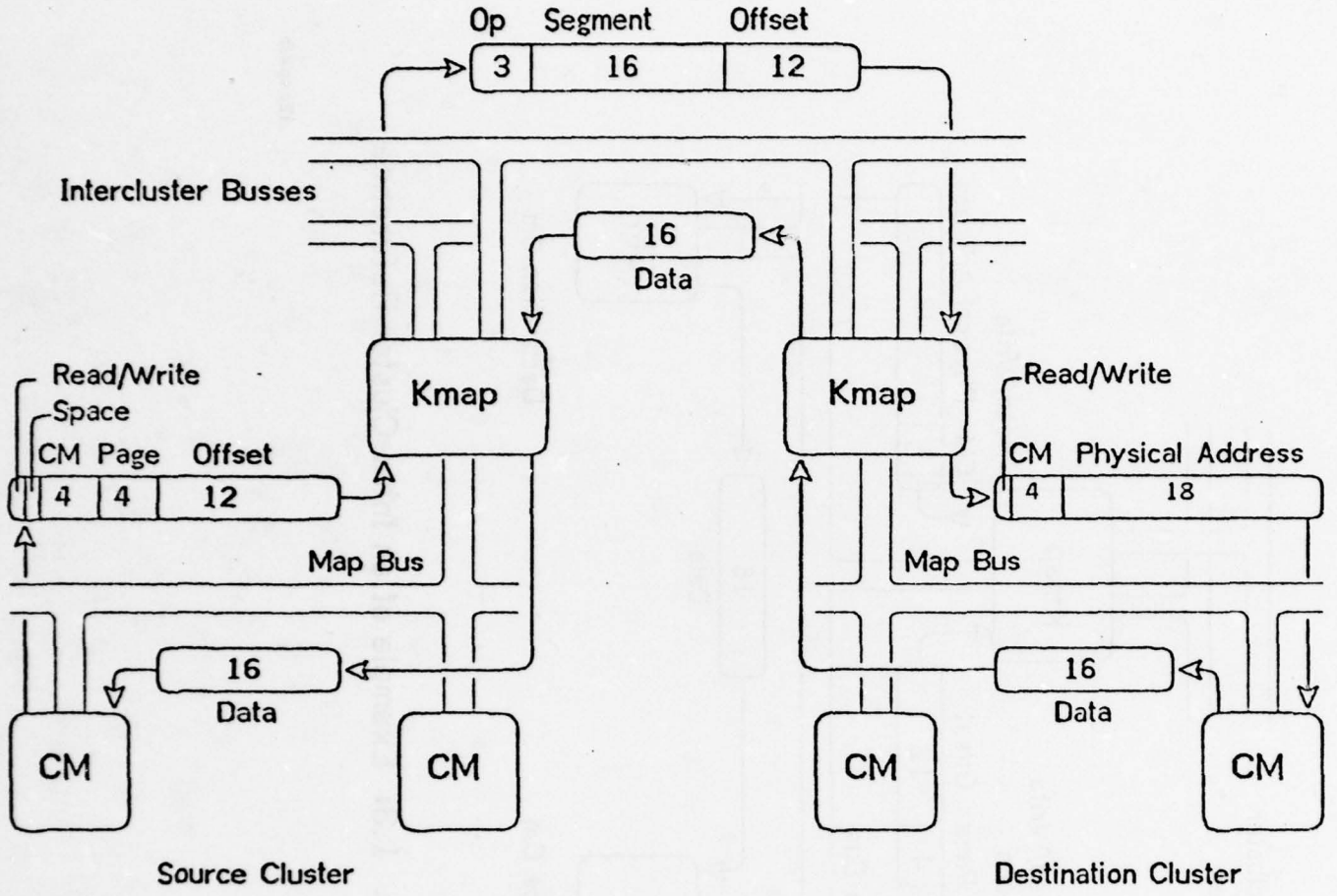


Fig. 1.4: Example of an Inter-Cluster Reference

t2e.spx/b

and the intercluster bus is now involved in the transferring of the packet between the clusters. To the processors and memories within the cluster the memory reference seems the same as for an intra-cluster reference.

In addition to the concurrency afforded in the mapping mechanism by having multiple clusters, the Kmap is partitioned into three units that allow pipelining of the communication mechanism within a cluster; a mapping processor (Pmap) is responsible for address translation and directs the actions of the two other components; a Map Bus controller (Kbus) is master of all the transactions of the asynchronous Map Bus and schedules activities for execution in the the Pmap; the third component (Linc) is responsible for shipping and receiving intercluster messages on the two intercluster buses to which each Kmap may connect. The three components are relatively independent and communicate via shared memory and a set of hardware queues.

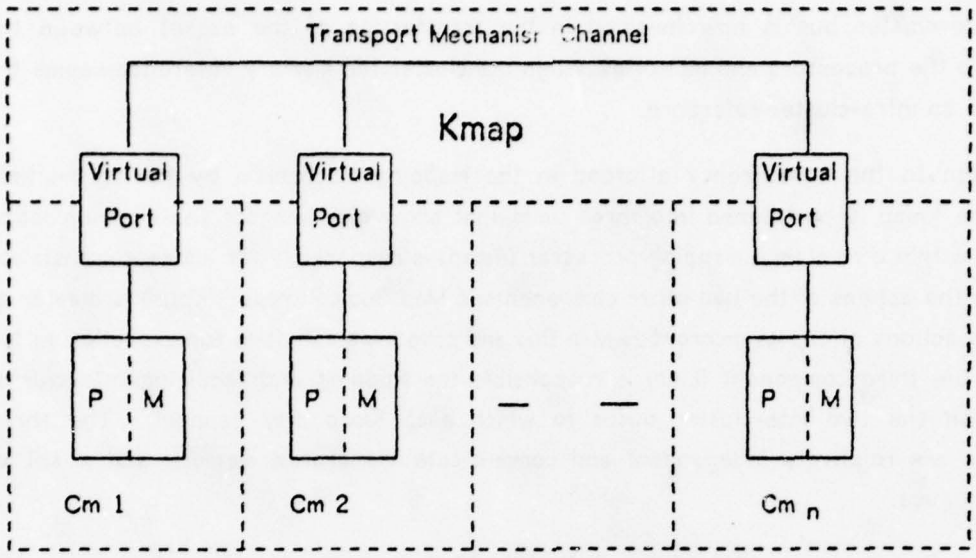
The initial Cm* system contained ten Cm's with 28 Kwords of memory each, which were configured into up to three clusters (three Kmaps were available). The next generation Cm* system is currently being built and will include 50 Cm's and five Kmaps.

1.3.1.2 Cm* as a local computer network system

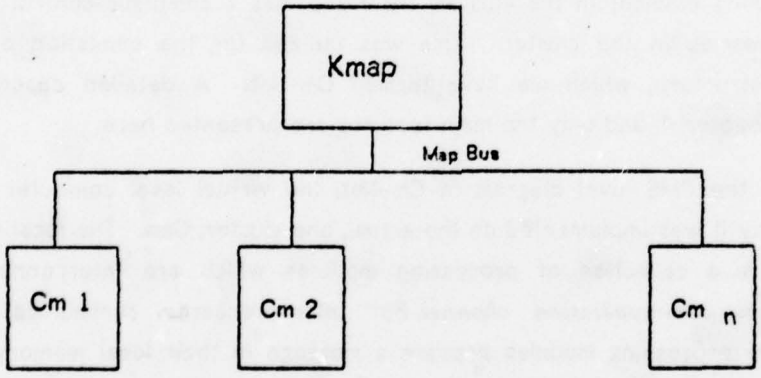
The central switching element in the cluster, the Kmap, has a complete control over the processors and memories in the cluster. This was utilized for the emulation of a local computer network structure, which we have termed Cm-Net. A detailed description of Cm-Net is given in chapter 4, and only the main features are presented here.

Figure 1.5 shows the PMS level diagram of Cm-Net; the virtual local computer network structure and the way it was implemented on the actual, one cluster, Cm*. The local computer network structure is a collection of processing modules which are interconnected via *transport mechanism communication channel*. For inter-processor communications the processors (P) in the processing modules prepare a message in their local memory (M). A controller, or *port*, in the sending site is responsible for retrieving the message from memory and transmitting it out into the channel, and for copying the message into the local memory in the receiving site. The messages are beeing transferred in packets of up to 256 words.

On the actual Cm* system most of the port functions and the transport mechanism channel were emulated by microprogramming the Kmap. This emulation is made so that the one cluster Cm* system, as shown in figure 1.5 (a), is seen by the user as the local network structure in figure 1.5 (b). This emulation required an addition of about 220 microinstructions to the initialization microprogram in the Kmap. The emulation used a DMA mechanism for a word by word transfer of the packet from one processor's memory to the other, without



b. The Virtual Hardware Architecture



a. The Actual Hardware Architecture

Fig. 1.5 Cm - Net, Cm* as a Local Computer Network

netwrl.apx/b

interfering with the processor's work.

A *process interface software support* package was written to simulate a simple network operating system. The package is a collection of application independent routines that supply the necessary functions that the application program might request, in an efficient manner. These functions include initialization of the network, interrupt handling, message buffers initialization and handling, sending and receipt of messages, timing and statistics, acknowledge mechanisms handling, etc. This package is replicated in each of the processing modules in the structure and executed by the processors in those modules.

1.3.2 Performance Models

Definitions [Svobodova 76]:

- A performance model is a sufficiently detailed abstraction of a system containing only the sufficient variables and relations needed for the performance evaluation. It is derived by analysis of a functional model for a specific model of workload.

- Functional model: Describes how the system operates. Defines the system such that it can be analyzed mathematically or studied empirically.

- Workload model: Represents system workload in a form required by the performance model - a controllable, reproducible environment.

- Performance measures: Set of predetermined parameters upon which the performance evaluation will be based (like throughput, utilization etc.).

The functional models that were constructed for the two multiple processor structures, were mainly at the PMS level of the structure. It is important to model the structural performance at this level for the system architect, the operating system builder, as well as for many potential users that need to know and utilize the hardware structure. Various models were constructed; The main technique used was the analytic queuing network technique, with several classes of customers.

The measurements results of a set of benchmark application programs, described in chapters two and four, constitute the workload model. Chapters three and four contain the description of the performance models that were constructed.

1.4 Synopsis of the Thesis

Chapter 2 includes the detailed discussion of the benchmark programs used and the measurements that were conducted on the Cm* system connected as a multiprocessor

structure. The main results are given in numerous tables and graphs.

Capter 3 contains the derivation of the various performance models used, their validation results and discussion of their respective merits. An example of applying these models for the investigation of performance issues in a multiprocessor structure is also included.

Chapter 4 includes a discussion of local networks, a detailed description of the emulation of Cm* as a local network, the protocols used, the benchmarks programs and their adaptation to a network environment, the main measurement results, the performance model developed and its main parameters, and a discussion of the merits of the structures investigated.

Chapter 5 concludes this thesis. It re-states the main results and indicates future research topics in the area of multiple computer structures.

The appendices contain summaries of the theoretical work and various proofs that were constructed during the derivation of the performance models.

2. Measurement and Evaluation of a Multiprocessor System

2.1 Introduction

Multiprocessor structures are just emerging; relatively little knowledge and experience, and practically no quantitative measurement results exist for these structures. In particular the performance evaluation of multiple computer-modules, i.e. pairs of processors and memories, demands specific performance measures (e.g. reference rate to code, data etc.). This chapter describes the result of applying the only practical method available to us at the present time - measuring the performance of the multiprocessor Cm* executing a set of benchmark programs.

Three programs from different application areas were used as the research vehicle to obtain these needed experience and measured data. This collection of programs was chosen as repreresentative of the major portion of the processing done for many scientific oriented applications. The programs differ enough to stress different aspects of the architecture (e.g. they exhibit different synchronization mechanisms, grain of parallelism, instruction set usage etc.).

These applications are[1]:

- Asynchronous iterative methods for solution of Partial Differential Equations (PDE's).
- Sorting (Quick Sort).
- Set Partitioning Integer Programming.

The main objectives of performing these measurements were:

1. Show the possibility of solving efficiently several problems on Cm* like structures (and in general in a multiprocessor environment) and thus the viability of such an architecture.
2. Obtain various performance measures to be used in the derivation and validation of computer-module performance models (which are discussed in Chapter 3) or for comparisons to other multiple processor structures (e.g. the computer network in Chapter 4).
3. Identify performance bottlenecks in the initial Cm* - 10 processor - hardware system that will provide some insight and help in tuning and constructing such systems in the future.

---

[1] Two other applications that were executed on Cm* are Harpy speech recognition system, running on the Staros operating system, and Algol 68 run time system. They are described in [Fuller 77], [Jones 78], [Hibbard 78]. Harpy is also discussed in section 4.3.

4. Provide some quantitative criteria to estimate decompositions of future applications for Cm* like structures into parallel tasks (both for the user and operating system design) and help to define classes of problems best suited for such a structure.

5. Show that a performance evaluation of a multiprocessor structure can be made using simple tools and techniques.

2.2 Techniques

2.2.1 Measurement Techniques

The measurements were made using both specially designed hardware and standard measuring equipment. Each Map Bus was attached to a Map Bus Monitor - a control panel plus specially designed logic which allowed a particular address or data value passing to and from a given computer-module to be displayed selectively and be counted. For example, the reference rate to a particular memory page could be counted.

A standard Logic Analyzer and counter were connected to the Kmap micro-instruction address lines and were used to monitor the Kmap micro-operations and determine what fraction of the Kmap time was spent in it's different operations.

2.2.2 Problem Decomposition Issues

Not much experience had been gained in the actual decomposition of problems into parallel tasks. The decomposition issue is not well understood yet and is currently a very active research area. For example, some of the last results (those that are mostly connected to the research at Carnegie-Mellon Univ.) include theoretical analysis of parallel algorithms [Kung 76]. Operating system design for multiprocessors [Wulf 74], [Jones 78]. Design aid tools used for algorithm decomposition [Peterson 77], [Brantley 77]. Programming languages design to assist the user in exploiting the parallelism (e.g. Algol 68, Concurrent Pascal) or even detect automatically the parallelism in the program written in a high level language and utilize the multiple processor structure to gain performance [Hibbard 78]. Some simple techniques that were used and proved to be effective (from the limited experience gained in decomposing these large grain, inherently parallel, applications), and some experience results are listed below:

1. Use of master/slave relationships between processes, where the master serves as the user interface process in the beginning and controls the other processes (slaves). The master also participates in the task as any other process during execution. Finally, the master outputs the task results when the work is done.

2. Use of a shared data structure as a pool from which all processes fetch the next task to be executed (when finished with their current task).

3. Partitioning a job into many small subtasks (a rule of thumb, probably a very loose one, used in the Integer Programming problem was ten times the number of processors) to be put in that shared data structure, thus minimizing the effect of distribution in subtask execution time and processors timing differences.

4. By exercizing care in the design and implementation of the application programs, and by using the relatively fast synchronization mechanisms supported by the Kmap - synchronization did not impose a severe performance degradation factor.

5. For the sake of debugging ease, it is very helpful to implement the program such that it may also be solved by only one (the master) process (which is of course very helpful for uni-processor comparisons).

6. For the relatively complex applications implemented, the amount of inherent, large grain, parallelism was surprisingly high.

2.3 The Benchmarks

This section contains a description of the three application programs that were implemented on the multiprocessor Cm*. For each program a description of the problem, the methods used to solve the problem, the implementation details and timing measurements - are given. Summary of the main results and their implications are given in the following section.

2.3.1 Numerical Application - Partial Differential Equations

2.3.1.1 The Problem

An example of an Asynchronous Iterative Method is the solution to Dirichlet's problem of Laplace's Partial Differential Equation (PDE) by the method of Finite Differences.

The program solves the PDE: $$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0$$

on a rectangular grid of size M x N, where only the values at the outer edges of the grid are given.

The Finite Difference method transforms the problem into a set of linear equations: $A\underline{x} = \underline{b}$, where $\underline{x}$ is an MN vector of all the points in the grid, A is an MN x MN sparse matrix and $\underline{b}$ is an MN vector derived from the boundary conditions. This set of linear equations is derived from the new approximate values of the points (in each iteration) by averaging the values of

the four adjacent neighbors of each point. The solution to this PDE is required in many application areas (e.g., Electro-Magnetic field, Hydrodynamics). Other PDE problems can be similarly solved using this method.

More details about Asynchronous Iterative Methods and their applications can be found in [Baudet 77, Baudet 78].

2.3.1.2 The Methods

Baudet [Baudet 78] gave a survey and developed several new methods for solving the above problem. Four of these methods were implemented and measured on Cm*. In all the methods, the computation is initially decomposed into P processes where P is equal to the number of processors available.. Each process (and processor) iterates on a fixed subset of MN/P components out of the total MN components. The four methods are briefly discussed below:

- Method 0: Jacobi's Method. In this method each processor retrieves its particular subset of the data from the global vector, $\underline{x}$, at the beginning of each iteration. New values are computed for the elements of $\underline{x}$ and then compared with their previous values. The elements are stored back into the global vector, $\underline{x}$, for other processors to use. This store operation is protected by a critical section. The processor then checks the error vector (computed by differencing the old and new values of the $\underline{x}$ vector). If the error vector is smaller than the pre-specified limit, the processor notifies the other processors that it has finished. If all of the other processors have finished their work the computation is complete. Otherwise the processor blocks awaiting the completion of the iteration by all the other processors before starting the next iteration.

- Method 1: Asynchronous Jacobi Method (AJ). This method is the same as the Jacobi method (Method 0) except that each processor does not wait for the other processors to finish before starting on the next iteration.

- Method 2: Asynchronous Gauss Seidel Method (AGS). This method is similar to AJ method (Method 1) except the processor uses the new values computed in its subset as soon as they are available (not the values known at the beginning of the iteration as in the previous two methods).

- Method 4: Purely Asynchronous Method (PA). This method computes a new value of each component using the most recent values of all components by reading them directly from the global vector, $\underline{x}$, and writing the updated values directly back to the global vector (without any critical sections or synchronization). This last method is clearly the most efficient. It also uses less memory than the other methods. It uses critical sections rarely to inform the master process that the work has been finished. Theoretically almost linear speed up can be achieved with this method.

More discussion of the above methods and experimental results on C.mmp (using floating

point) can be found in [Baudet 1978]. Fixed point, single precision computation was used in the Cm* implementation.

Some features of this problem:

- Extensive use of integer arithmetic operations.
- Four different methods with different synchronization requirements.
- Almost linear speed up can be expected.

2.3.1.3 The Implementation

The grid size was chosen to be 21 x 24 points, i.e., a linear system of 504 elements. The boundary conditions were chosen to be all zeroes and the grid points initialized to one. The error bound was chosen to be 0.1 in all the following experiments. One processor, the *master* processor, initializes and starts the other *slave* processors, and prints the results when all have finished. Note that the *master* participates in the computation like any other (*slave*) processor. All the global variables are kept in the *master* processor's local memory area. Synchronization and mapping are achieved by using the *simple* Kmap micro-code.

2.3.1.4 The Measurements

The timing measurements and speed up factors for various memory reference patterns are given in Figures 2.1 to 2.8 and in Table 2.1. The table shows the timing results for six different memory patterns executed by a single processor:

- All local - all the memory references are directed from the processor to its local memory in the Cm.
- All mapped - all the memory references are mapped to the Kmap and returned to the Cm's local memory.
- Only code mapped - only the memory references to the application's code are mapped via the Kmap (the rest are directly sent to the local memory).
- Only stack mapped - only the memory references to the processor's stack area are mapped via the Kmap.
- Only Local variables mapped - only the memory references to the variables which are local to a process (those that are not shared between processes) are mapped via the Kmap.
- Only global variables mapped - only the memory references that are shared between cooperating processes are mapped via the Kmap.

Table 2.1: PDE, Memory Reference Patterns (using one Cm)

Execution times (in seconds) in various memory reference patterns and percentage of time it takes to execute the task compared to the time to execute it when all memory references are local to the same memory in that Cm.

Method 0

Memory reference pattern	Execution time	% Local execution time
All Local	362	100
All Mapped	954	263.5
Only Code Mapped	835.5	231
Only Stack Mapped	420	116
Only Local Variables Mapped	405	112
Only Global Variables Mapped	378	104.5

Method 1

Memory Reference Pattern	Execution Time	% Local execution time
All Local	355.5	100
All Mapped	948	267
Only Code Mapped	820	231
Only Stack Mapped	413	116
Only Local Variables Mapped	399	112
Only Global Variables Mapped	370	104

Method 2

Memory Reference Pattern	Execution Time	% Local execution time
All Local	181	100
All Mapped	478	264
Only Code Mapped	417	230
Only Stack Mapped	210	116
Only Local Variables Mapped	203.5	112
Only Global Variables Mapped	188	104

Method 4

Memory Reference Pattern	Execution Time	% Local execution time
All Local	165.5	100
All Mapped	433	261.5
Only Code Mapped	382	231
Only Stack Mapped	196	118.5
Only Local Variables Mapped	176.5	107
Only Global Variables Mapped	173	104.5

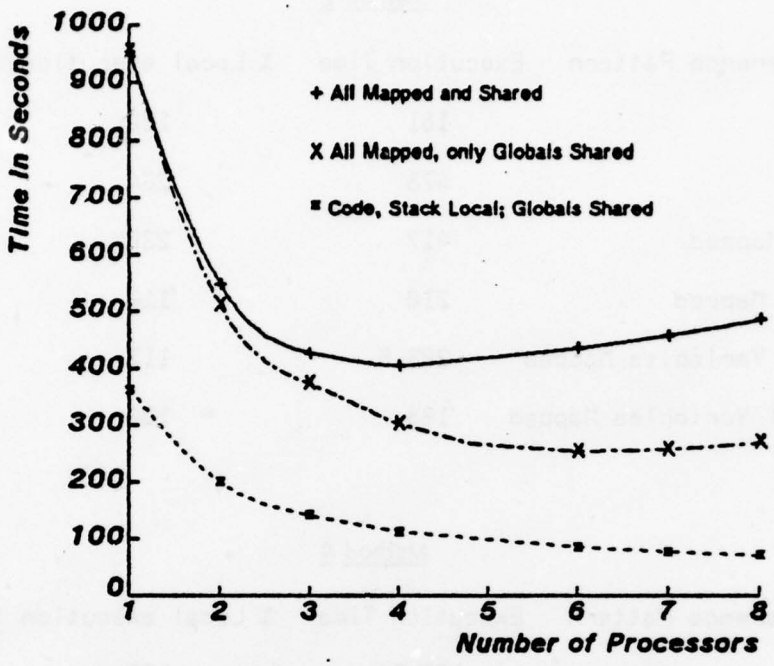


Fig. 2.1: PDE, Method 0, Execution Time

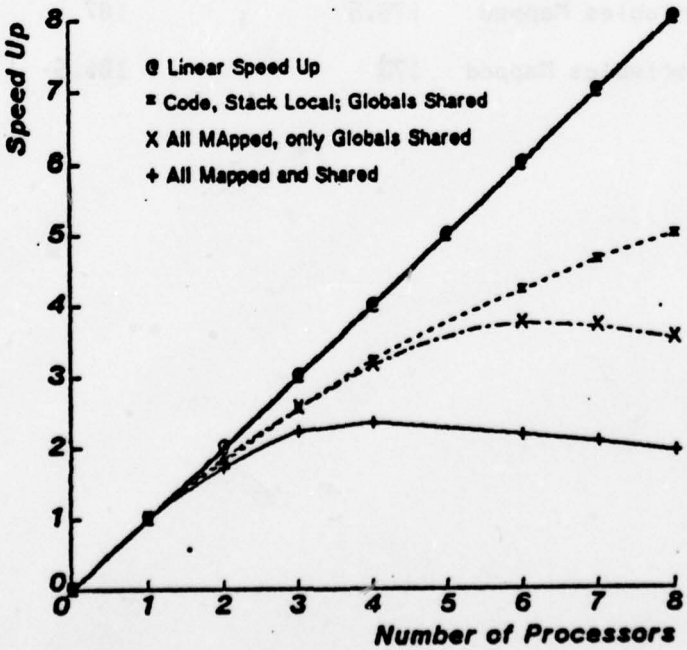


Fig. 2.2: PDE, Method 0, Speed Up

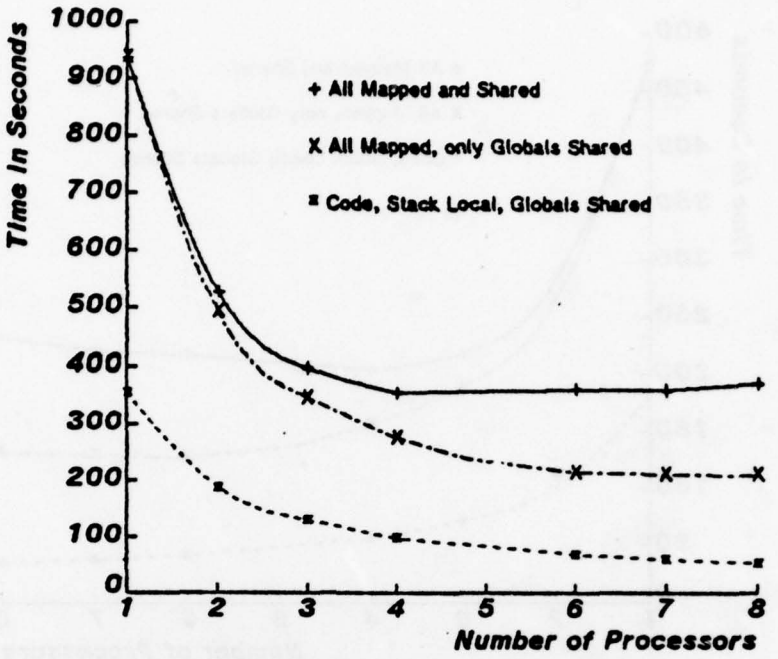


Fig. 2.3: PDE, Method 1, Execution Time

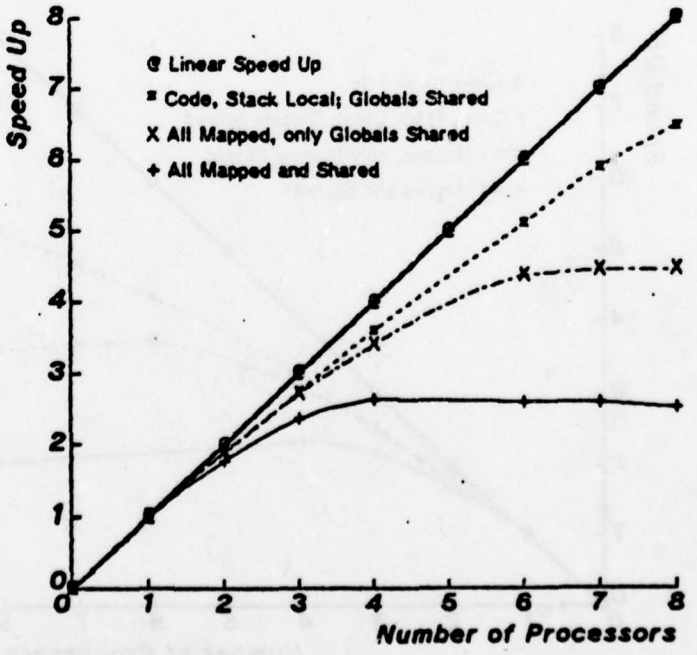


Fig. 2.4: PDE, Method 1, Speed Up

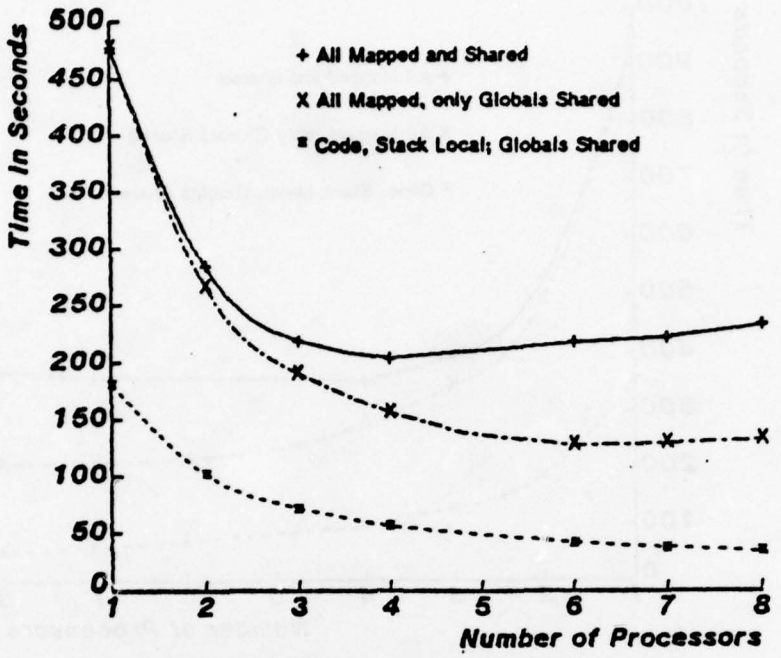


Fig. 2.5: PDE, MEthod 2, Execution Time

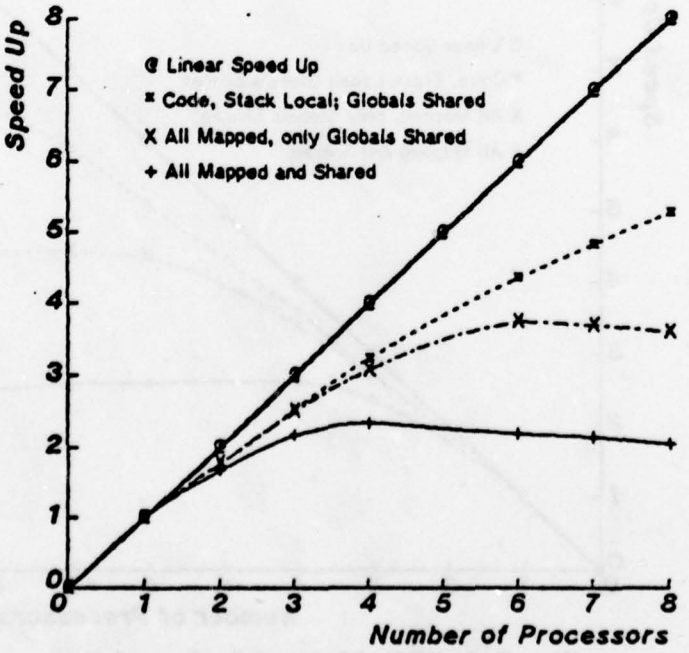


Fig. 2.6: PDE, Method 2, Speed Up

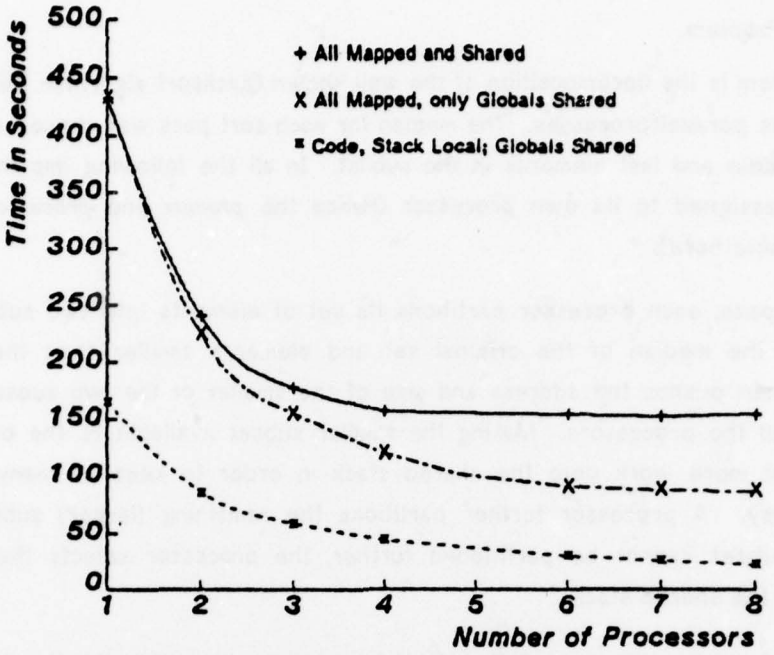


Fig. 2.7: PDE, Method 4, Execution Time

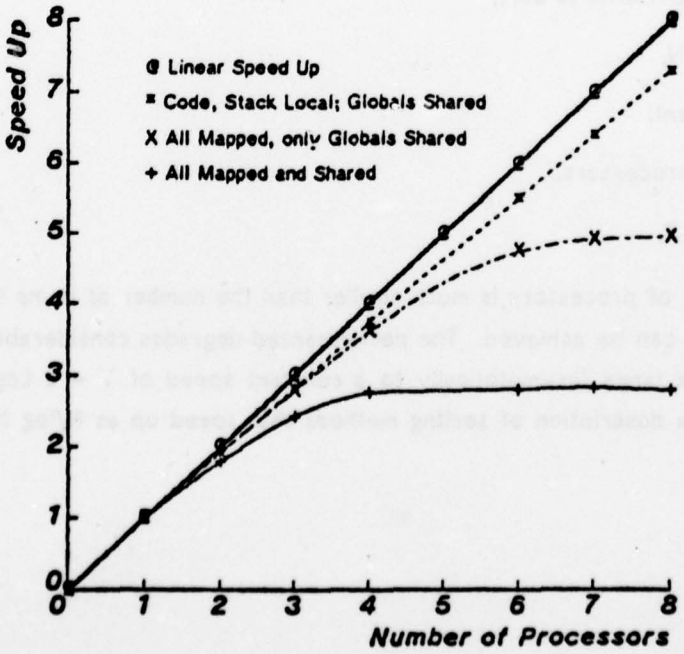


Fig. 2.8: PDE, Method 4, Speed Up

2.3.2 Sorting - Quick Sort

2.3.2.1 The Problem

This problem is the decomposition of the well known Quicksort algorithm [Singleton 69] into asynchronous parallel processes. The median for each sort pass was chosen as the median of the first, middle and last elements in the sublist. In all the following implementations, each process is assigned to its own processor (Hence the *process* and *processor* may be used interchangeable here).

During a pass, each processor partitions its set of elements into two subsets: Elements larger than the median of the original set and elements smaller than the median. The processor then pushes the address and size of the smaller of the two subsets onto a stack shared by all the processors. Making the smaller subset available to the other processors tends to put more work onto the shared stack in order to keep as many processors as possible, busy. A processor further partitions the remaining (larger) subset. When the remaining subset cannot be partitioned further, the processor selects the next available subset from the shared stack.

Very simple assumptions about the algorithm (similar assumptions leading to $T = c\ N \text{ Log } N$ for sorting using the sequential algorithm) give a theoretical sorting time of:

$$T = c\ N\ \{ (K-M)/P + 2(1 - (1/2)^M) \}$$

Where:

- N = # of elements to sort,
- $K = \text{Log}_2 N$,
- c = constant,
- P = # of processors,
- $M = \text{Log}_2 P$,

If the number of processors is much smaller than the number of items to be sorted, almost linear speed-up can be achieved. The performanced degrades considerably when the number of processors is large (asymptotically to a constant speed of $T = c \text{ Log } N/2$). See Stone [Stone 71] for a description of sorting methods that speed up as N/log N for large numbers of processors.

2.3.2.2 The Implementation

One processor, the *master* processor, initializes and starts the other processors. It makes the first partition and prints the results when the sort is complete. The *master* also participates in the sort like any other (*slave*) processor.

The stack, the vector of elements to be sorted, and the global variables are kept in the local memory of the *master* processor. All the experiments sort 18,000 elements, where each element is a 16 bit (2's complement) value.

Some features of this problem:

- Extensive access to the shared data vector (which causes data contention).
- Extensive use of logical operations.
- Almost linear speed up when the number of elements in the data part is orders of magnitude larger than the number of processors.

2.3.2.3 The Measurements

The timing measurements and speed up factors for various memory reference patterns are given in Figures 2.9, 2.10 and Table 2.2.

Table 2.2. Quicksort, Memory Reference Patterns (using one Cm)

Execution times (in seconds) in various memory reference patterns and percentage of time it takes to execute the task compared to the time to execute it when all memory references are local to the same memory in that Cm.

Memory Reference Pattern	Execution time	% local execution time
All Local	25.54	100
All Mapped	70.0	274
Only Code Mapped	58.4	229
Only Stack Mapped	30.3	118.5
Only Local Variables Mapped	28.3	111
Only Global Variables Mapped	31.2	122

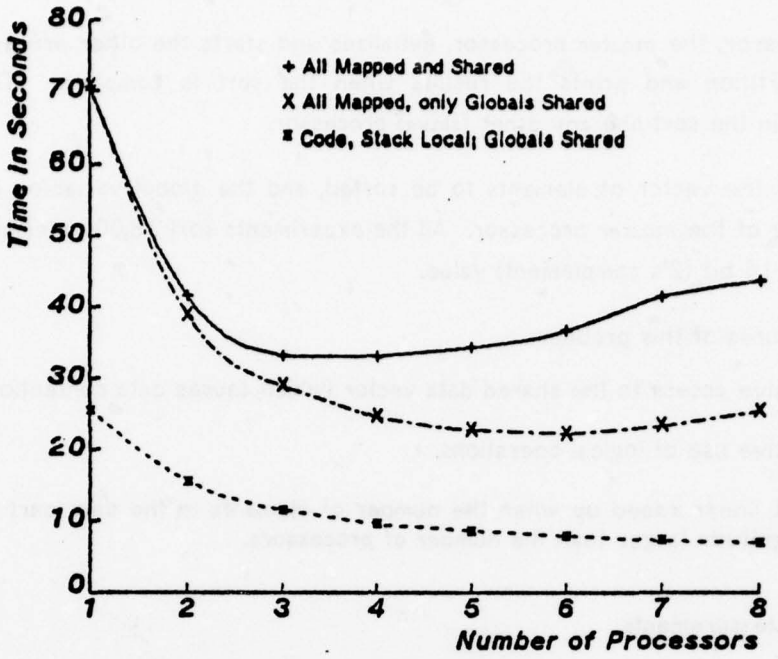


Fig. 2.9: Quick Sort, Execution Time

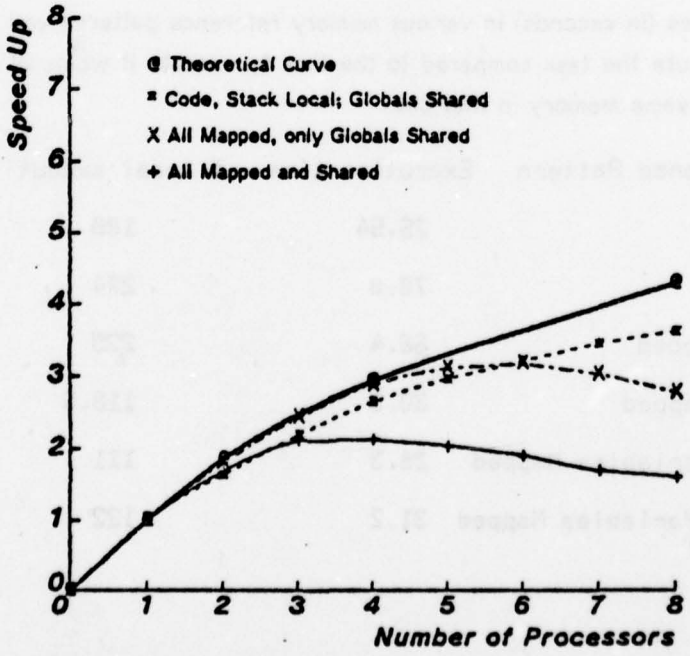


Fig. 2.10: Quick Sort, Speed Up

2.3.3 Searching - Set Partitioning Integer Programming

2.3.3.1 The Problem

The particular integer programming solution considered here is one of the most practical methods. It is used, for example in airline crew scheduling [Balas 76].

This problem is typically solved with an enumeration algorithm, by searching (N-ary tree search) in a large, relatively sparse, binary matrix - typically on the order of hundreds by thousands - for a minimum cost solution.

The set partitioning problem is to solve:

$\min \{ \underline{c}.\underline{x} \mid A\underline{x} = \underline{e}, x_j = 0 \text{ or } 1 \text{ for } 0 \le j \le N \}$

Where:

- A = M x N binary matrix.
- $\underline{c}$ = N vector
- $\underline{e}$ = (1.....1) M vector

As an example of this method, consider the airline crew scheduling problem. The rows of the A matrix correspond to a set of flight legs (from city A to city B, in time T) to be covered during a specified period and the columns of A correspond to a possible sequence of tours of flight legs done by one crew, $\underline{c}$ is the vector of associated cost of each tour. A possible solution includes a set of tours that satisfy all the flight legs (one and only one crew makes a flight leg). We are looking for the solution with the lowest cost.

Some features of this program:

- It uses binary data types, manipulates a large matrix while requiring a relatively small address space.
- Extensive use is made of both arithmetic and logic functions.
- There is a theoretical possibility of nearly linear speed up.

2.3.3.2 The Implementation

As in the previous applications, one processor - the *master* - initializes, creates the array according to user's specification and puts enough initial possible search path solutions in a global stack, from which all the processors pick their work. The criteria was (arbitrarily

chosen) to put more than 10 x P path solutions into the stack (where P = number of processors) - so that work will be more evenly distributed between the processors and all will be occupied a large percentage of the time.

To enhance pruning in the search, a global variable contains the cost of the best solution found so far by any of the processors - and all compare their current cost value to it and begin to track back in the search when that global cost is lower.

2.3.3.3 The Measurements

Five different cases where arbitrarily chosen as test cases.

-	I	J	Seed	Density	Number of Solutions
Case 1	10	100	1	0.1	3
Case 2	10	100	2	0.1	5
Case 3	10	100	3	0.1	5
Case 4	50	500	1	0.2	0
Case 5	17	60	1	0.1	1

Where:

- I = number of rows in the A matrix
- J = number of columns in the A matrix
- Seed =Initial seed number for a random number generator to generate the matrix
- Density =Density (ratio of ones and zeroes) in the array
- Number of Solutions = number of different solutions found by one processor

The timing, speed up factor and the number of nodes examined for the 5 different cases are given in Figures 2.11 to 2.13 and in Table 2.3. Note the interesting fact that almost a constant number of nodes are examined while the number of processes participating in the task varied.

Table 2.3: Integer Programming, Memory Reference Patterns (one Cm)

Execution times (in seconds) in various memory reference patterns and percentage of time it takes to execute the task compared to the time to execute it when all memory references are local to the same memory in that Cm. Three cases are presented here.

Case 1

Memory reference Pattern	Execution Time	% Local Execution Time
All Local	79.1	100
All Mapped	215.1	272
Only Code Mapped	176.8	223.5
Only Stack Mapped	113.9	143
Only Local Variables Mapped	85.8	108.5
Only Global Variables Mapped	81.1	102.5

Case 2

Memory reference Pattern	Execution Time	% Local Execution Time
All Local	19.5	100
All Mapped	53.0	272
Only Code Mapped	43.6	223.5
Only Stack Mapped	27.3	140
Only Local Variables Mapped	21.1	108.2
Only Global Variables Mapped	20.0	102.5

Case 4

Memory reference Pattern	Execution Time	% Local Execution Time
All Local	204.4	100
All Mapped	546.1	267
Only Code Mapped	455.5	223
Only Stack Mapped	277.4	136
Only Local Variables Mapped	217.5	106.5
Only Global Variables Mapped	208.3	102

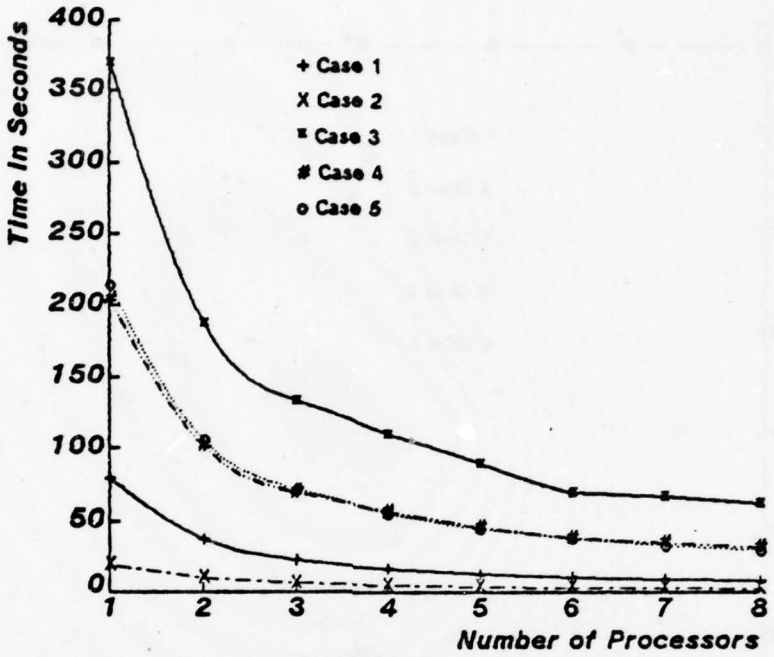


Fig. 2.11: Integer Programming, Execution Time

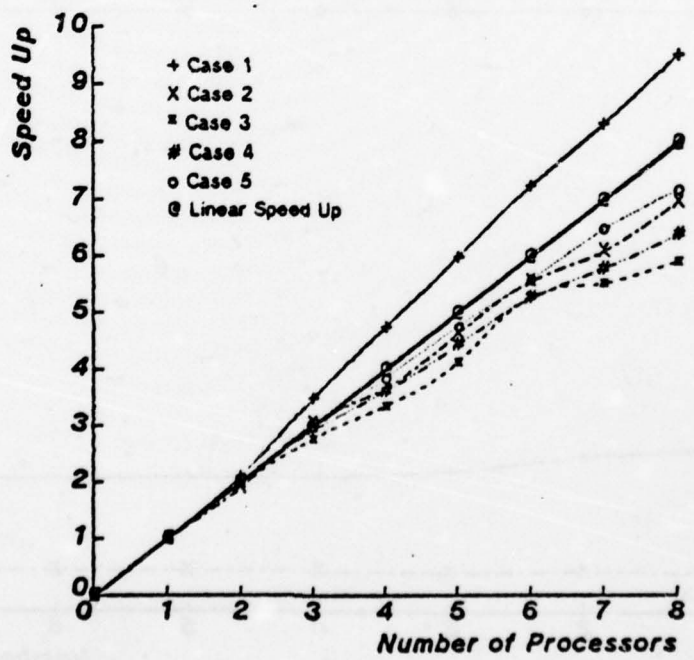


Fig. 2.12: Integer Programming, Speed Up

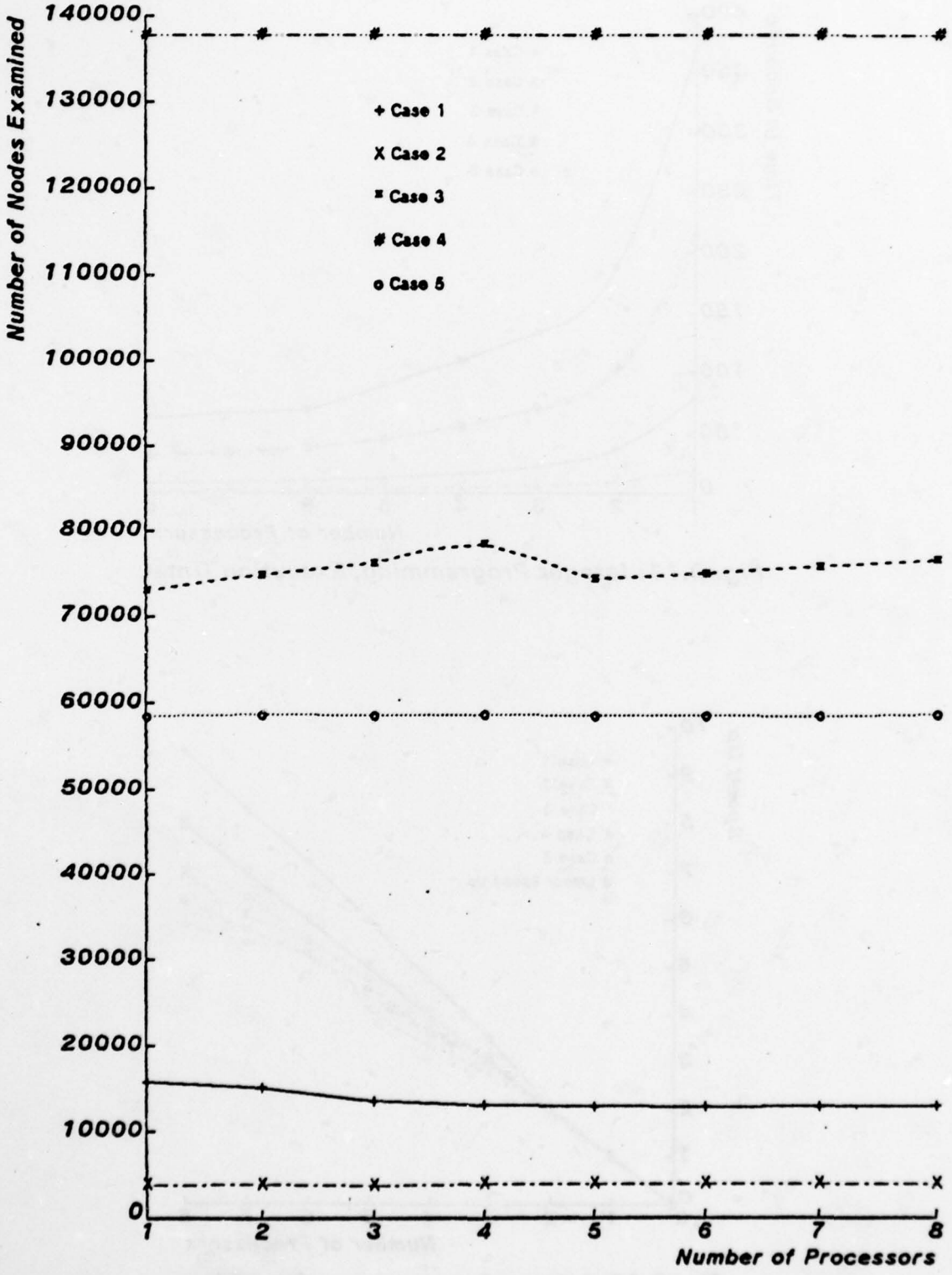


Fig. 2.13: Number of Nodes Examined in Integer Programming

2.4 Summary of Measurement Results for a One Cluster System

This section summarizes the results of the tables and figures presented in the previous section. It shows the main interference and degradation effects of the one cluster system.

2.4.1 Access Times in the Cm* System

The following summarizes the results presented in Tables 2.1 to 2.3. The ratio figure given, is the ratio of total execution time (for various memory reference patterns) to the total execution time when all memory references are to local memory.

- When all references are mapped, the ratio is between 2.6 and 2.75.
- When code is mapped and everything else is local, the ratio is between 2.2 and 2.3. Hence, it is very important for code to be in a Cm's local memory.
- When the Cm's stack is mapped, the ratio is between 1.16 and 1.185 for the PDE and QuickSort applications and 1.4 for the Integer Programming application. The latter application consists of a large number of small routines, the execution of which causes frequent stack accesses to perform the call/return sequence.
- When global data is mapped, the ratio varies between 1.02, when global data accesses are infrequent, and 1.15 when accesses are relatively frequent. These particular figures are encouraging since large shared data structures may be located anywhere in the system without significant performance degradation.
- When own (local) data is mapped, the ratio is between 1.07 and 1.12.

These results show the importance of localizing code in the Cm's local memory, while global data can be put anywhere in the cluster without severe performance degradation.

2.4.2 Throughput of Cm* Buses and Components

This subsection shows the way interference effects can limit the number of processors which are effectively utilized in a one cluster structure.

- When all processors share both code and data from a single memory, the graphs indicate that performance cannot be improved by using more than 3 or 4 processors. This limitation is caused by memory contention.
- The graphs showing Cm's making references which are mapped back to their local memory ("Mapped to Self") indicate that the Kmap saturates when 6 or 7 processors are simultaneously active in this mode.

2.4.3 Hit and references ratios

The rate of mapped memory references from a Cm, for which *all* references were mapped, was measured. The reference rates to Code, Stack, Own (local data to a process) and Global data were also measured. The percentages of the total reference rate represented by the above reference types are tabulated below. The measurements were made using a combination of the Map Bus Monitor and a frequency counter.

PDE

	method 0	method 1	method 2	method 4
Code	80.8%	80%	80.5%	82%
Stack	10%	10%	10%	11.5%
Owns	6.8%	7%	7%	4%
Globals	2.5%	3%	2.5%	2.5%

Quick sort

Code	71%
Stack	12.5%
Owns	6.5%
Globals	9.5%

Integer programming

-	case 1	case 2	case 3	case 4	case 5
Code	71.3%	70.3%	71.1%	72.8%	71.5%
Stack	23%	24%	25.4%	22.3%	23.6%
Owns	4.75%	4.6%	4.15%	3.75%	3.85%
Globals	1.1%	1.1%	1.2%	1.1%	1.1%

The hit ratios (ratio of references to local memory to the total memory references - when only global variables are mapped) are therefore on the order of 97.5% in the PDE program, 90.5% in the Quicksort program and 99% in the Integer Programming program.

2.4.4 Utilization

2.4.4.1 System components

Figures 2.14 to 2.17 show the different aspects of the system's component utilization while running the PDE programs. The Kmap executes a two micro-instruction loop (called the "idle loop") whenever there is no useful work for it to do (This is used to show the utilization of the Kmap). The "Slocal busy loop" is seven micro-instructions long and is used to show the

amount of contention for Slocals and memories. This loop is executed when the Kmap tries to access a memory but must wait because the Slocal is busy executing a previous reference. A successful reference to an Slocal takes four micro-instructions.

The Kmap micro instruction time is about 157 nanosec.

It is interesting to note that the maximum Pmap utilization when the hit ratio is 0%, i.e. all references are mapped back to the Cm, is only 35%. The rest of the time is spent in the idle loop.

2.4.4.2 Number of Cms supported by a Kmap

The measurements shown in Fig. 2.15 (all references mapped) show that, due to Kmap contention, there is a degradation of about 10% in the time to execute a remote (mapped) reference inside the cluster when about 400,000 references/sec. are made to the Kmap.

Consider, for example, the PDE program. The average number of mapped references from one Cm, when only global variables are mapped, is about 6,500 references/sec. This means that the Kmap can support 400,000/6,500 or about 60 Cm's with only 10% degradation in the mapped reference time. With a 90% hit ratio (10% mapped references), as observed in the Quicksort program, the Kmap can support about 20 processors with a 10% degradation in the mapped reference time.

2.4.5 Total Local Memory References per Second and MIPS.

Based on the three programs measured, the local memory reference rate was as follows:

- PDE -Time between successive memory references = 3.7 microseconds. Reference rate = 270 KWords/Second.
- Quick Sort -Time between successive memory references = 3.33 microseconds. Reference rate = 300 KWords/Second.
- Integer Programming -Time betwen successive memory references = 3.51 microseconds. Reference rate = 285 KWords/Second.

The average number of memory references per instruction was measured to be:

- QUICKSORT - 1.75
- PDE - 1.45
- INTEGER PROGRAMMING - 1.75

For the HARPY speech recognition system the reference rates are in the 1.85 - 1.9 range. (See [Fuller 77b].)

These numbers are considerably lower than the corresponding numbers (2.1 to 2.3) measured on a large sample (about 8 million instructions) of 20 Fortran, Cobol and operating system programs [Marathe 77]. A possible explanation for this may be that the Cm* applications were compiled by a good optimizing compiler which effectively uses the general purpose registers available, and thus minimizes accesses to memory.

The average number of memory references per instruction in the measured programs was about 1.7. From this we get that the maximum potential instruction rate of a Cm executing a program in its local memory is on the order of 0.17 MIPS (Millions of Instructions Per Second). This may be extrapolated to 1.7 MIPS for a 10 processor system.

2.4.6 Memory Contention

As seen in Fig. 2.17 (Slocal busy loop count) the time added to a reference by each Slocal busy loop count is about 2.26 microseconds (1.1 microseconds in the Pmap and 1.16 microseconds in the Kbus and Map bus).

2.4.6.1 PDE

The number of busy loop references in the PDE program increases from 0 to 4200 in Method 0 and to 9600 in Method 4 - with 0 to 8 Cm's all running local code (and shared data). This means that, as expected, the degradation due to memory contention in this high hit ratio application is negligible. The memory contention adds only 0.5% - 1% to the utilization of the Pmap. (The Pmap spends 2% - 3% of its time doing useful work, and sits in its idle loop for the remaining 96% - 97% of the time.)

From the above results, the performance degradation due to memory contention (and hence slower mapping) was calculated to be in the range: 0.1% to 0.25%.

2.4.6.2 Quick Sort

This program is difficult to measure as it has a short execution time and changes its demands upon the Kmap during execution. Taking peak Kmap activity with 8 processors (local code):

- Slocal busy loop 100,000 counts (11.0% of Pmap time)
- Idle loop 2,500,000 counts (78.5% of Pmap time)

- Useful mapping references 150,000 counts (10.5% of Pmap time)

From the above results, the performance degradation due to memory contention was calculated to be 2.7%. Map bus contention was ignored in the above calculations.

2.4.7 Frequency of Lock operations

The total number of lock operations per second ranged, in PDE Method 0, from 10 in a 2 Cm system to 3,400 for the 8 Cm system. For Quick Sort the range is 1,500 - 14,000. These are algorithm dependent numbers and do not have much influence on total execution time due to the fast synchronization operation in the simple Kmap microcode.

2.4.8 Timing Delays

Tables 2.4 and 2.5 contain the detailed timing information concerning the various delays that a memory reference sufferes when executing an intra-cluster or an inter-cluster memory reference respectively. These timing delays are for a system without contention (i.e. when the reference is the only one in the system) and was obtained by a combination of direct timing measurements and Kmap micro-code counts.

Table 2.4: Timing of an Intra-Cluster Reference

The following timing is for a intra cluster read request when there is no contention on the Kmap.

Timing (in μsec):

- CPU avg. processing time (assuming 3.5 μsec cycle time, 1.6 μsec in the Mem. bus) ---------- 1.9
- CPU to Kbus request time ---------- 0.9
- Kbus to Pmap processing time ---------- 0.523
- Waiting in Kbus to Pmap queue ---------- 0.3
- Processing time in Pmap ---------- 0.628
- Waiting in Pmap to Kbus queue ---------- 0.3
- Kbus to Slocal processing time ---------- 0.314
- DMA arbitration to get map bus ---------- 0.8

- DMA memory access time ---------- 0.9
- Slocal to Kbus return request time ---------- 0.7
- Kbus to Slocal return request processing time ---------- 0.942
- Kbus to CPU return request ---------- 1.2

- TOTAL ---------- 9.407

Table 2.5: Timing of an Inter-Cluster Reference

The following timing is for an inter cluster read request when there is no contention.

Timing (in μsec):

- CPU avg. processing time (assuming 3.5 μsec cycle time, 1.6 μsec in the Mem. bus) ---------- 1.9
- CPU to Kbus1 request time ---------- 0.9
- Kbus1 to Pmap1 processing time ---------- 0.52
- Waiting in Kbus1 to Pmap1 queue ---------- 0.3
- Processing time in Pmap1 (Send req.) ---------- 2.83
- IC bus busy time ---------- 1.8
- Fix (various reasons) Linc delay ---------- 0.9
- Linc to Kbus1 delay ---------- 0.9
- Kbus2 processing (to Pmap2) ---------- 0.21
- Kbus2 to Pmap2 queue ---------- 0.3
- Pmap2 Ic (receive) processing time ---------- 1.73
- Waiting in Pmap2 to Kbus2 queue ---------- 0.3
- Kbus2 to Slocal processing time ---------- 0.31

- DMA arbitration to get map bus ---------- 0.8
- DMA memory access time ---------- 0.9
- Slocal to Kbus2 return request time ---------- 0.7
- Kbus2 to Pmap2 return request processing time ---------- 0.52
- Kbus2 to Pmap2 queue ---------- 0.3
- Pmap2 IC send return req. ---------- 1.1
- IC bus busy time (return req.) ---------- 1.4
- Linc delay (various reasons) ---------- 0.9
- Linc to Kbus1 delay ---------- 0.9
- Kbus1 to Pmap1 processing ---------- 0.21
- Kbus1 to Pmap1 queue ---------- 0.3
- Pmap1 processing (return req.) ---------- 0.63
- Pmap1 to Kbus1 queue ---------- 0.3
- Kbus1 (return) processing ---------- 0.84
- Kbus1 to CPU return request ---------- 1.2

- TOTAL ---------- 23.9

2.4.9 Some Useful Numbers

Some rounded, useful numbers from this evaluation of Cm* include:

- 1 Cm (i.e. single LSI-11 processor) - 0.17 MIPS
- Reference saturation rate of shared memory bus - 270 KWords/Second (3.7 microsec per reference).
- Reference saturation rate of Kmap (including the map bus) - 552 KWords/Second.
- Saturation of map bus transactions (i.e read or Read-Modify-Write between two Cm's needs three transactions and write requires two transactions) - 1.7 MWords/Second.

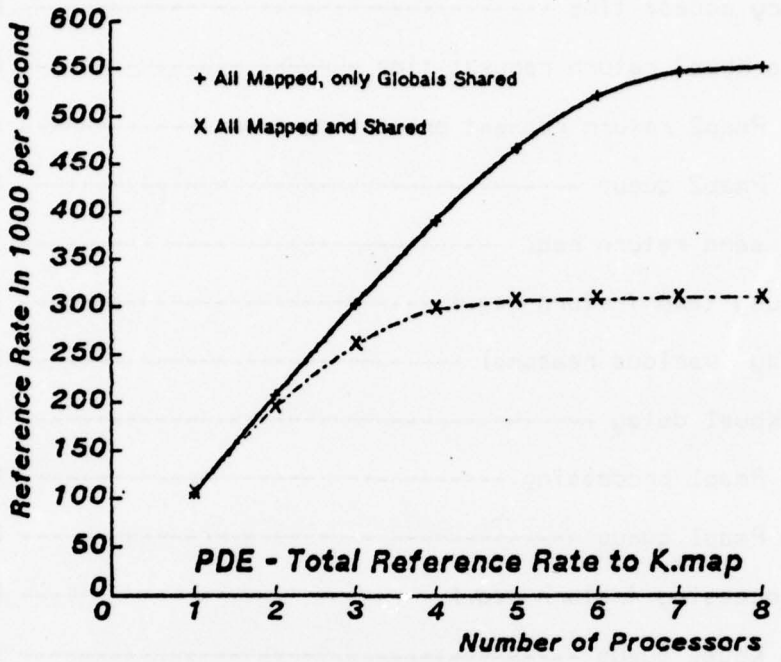


Fig. 2.14: Utilization of Buses

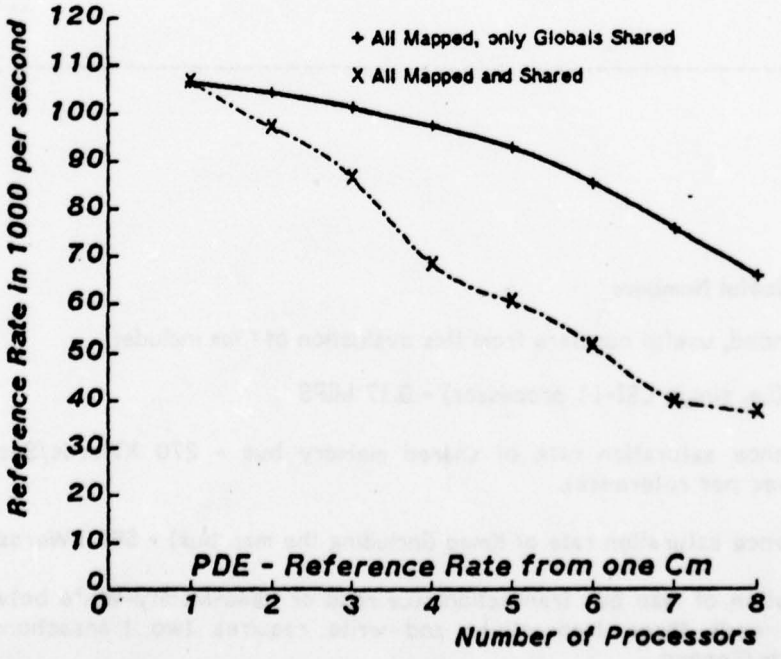


Fig. 2.15: Utilization of Processors

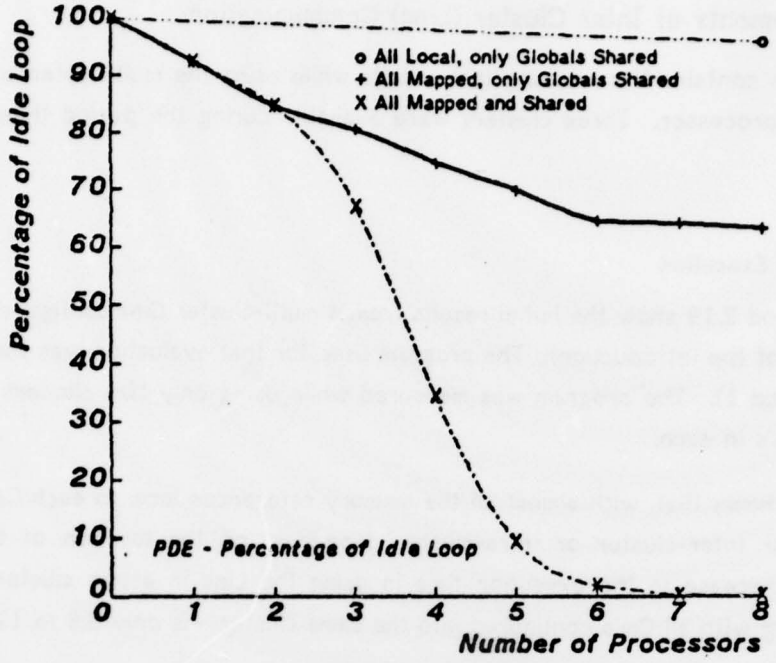


Fig. 2.16: Utilization of Kmap

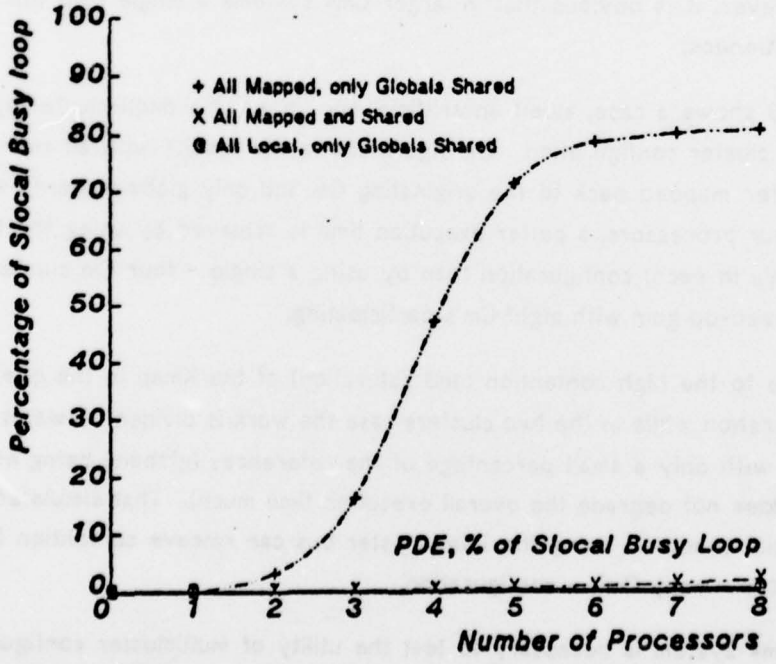


Fig. 2.17: Utilization of Kmap (b)

2.5 Measurements of Inter Cluster (Linc) Communication

This section contains the measurement results while using the multicluster configuration of the Cm* multiprocessor. Three clusters were available during the period these experiments were made.

2.5.1 Program Execution

Fig. 2.18 and 2.19 show the initial results from a multi-cluster Cm* configuration (as shown in Figure 1.2 of the introduction). The program used for that evaluation was the PDE program (methods 0 and 1). The program was measured while using only two clusters with an equal number of Cm's in each.

Fig. 2.18 shows that, with almost all the memory references local to each Cm (only globals mapped either inter-cluster or intra-cluster depending on the location of the Cm in the system) the increase in the execution time in using the Linc in a two cluster configuration over executing with all Cm's configured into the same cluster - is only 5.5 to 12%.

Although the multi-cluster performance of Figure 2.18 is not much worse than the single cluster configuration, it does raise the issue of when, if ever, it makes sense to use a multicluster configuration. With only 10 Cm's in the initial configuration we did not observe any practical situation in which multi-cluster Cm* configurations were superior to a single cluster. However, it is obvious that in larger Cm* systems a single Map bus and Kmap will become a bottleneck.

Figure 2.19 shows a case, albeit an artificial one, in which a multi-cluster system is better than a single cluster configuration. The figure shows the results with all memory references within a cluster mapped back to the originating Cm and only globals shared across clusters. Starting in four processors, a better execution time is achieved by using the Linc in a a two cluster (2 Cm's in each) configuration than by using a single - four Cm cluster; increasing to about 36% speed-up gain with eight Cm's participating.

This is due to the high contention (and saturation) of the Kmap in the one cluster (0% hit ratio) configuration while in the two clusters case the work is divided between the two Kmaps participating with only a small percentage of the references (globals) being executed via the Linc (which does not degrade the overall execution time much). That simulates a case where, in some special condition, using the Inter cluster bus can releave contention (and saturation) in a one cluster - many Cm's - configuration.

A large Cm* system is necessary to test the utility of multicluster configurations running

practical applications.

2.5.2 Contention

Fig. 2.20 and 2.21 show the results of executing a simple, one instruction loop (L: mov *L, R7), test case through the Linc. The system was in a two cluster (eight and two Cm's in each) configuration, with all references initiated in the source (eight Cm's) cluster reading from memory in the other cluster.

Fig. 2.20 shows the reference rate change in the source cluster with increasing number of processors participating. The Figure shows different memory access methods:

- sharing the same memory in the destination cluster, or alternating between the memories of two Cm's.
- using a single Linc, or alternating between two Lincs (ports).

Fig. 2.21 shows the utilization of the source Pmap while executing the above cases. Utilization was measured as the rate of Kmap Idle Loop execution.

From these graphs a few observations and parameters can be deduced:

- The time to execute an inter-cluster reference (without contention in the system) is 26.4 μseconds.
- With only one processor executing, the source Pmap is busy only 13.2% of the time - i.e about 3.5 μseconds of the inter-cluster reference time is spend in the source Pmap (the same as received from a theoretical calculation based on the number of micro-instructions executed). Similar measurements of the destination cluster shows it being busy 10.3% or about 2.7 μseconds per reference.
- From Figure 2.20, we see that a single Linc has a bandwidth of about 200,000 references per second. When multi-Lincs are used, and now the source Pmap becomes the bottleneck, we do not reach saturation using 8 Cm's. Extrapolating from the utilization of the Pmap in Figure 2.21 (and from the 3.5 micro second execution time per reference of the source Pmap) the inter-cluster saturation rate is estimated to be about 287,000 references/second.
- The anomalous behavior of some of the graphs in Fig. 2.20 and 2.21, when severe contention occurs in the 7 to 8 Cm's region, is not well understood yet and needs further investigation.

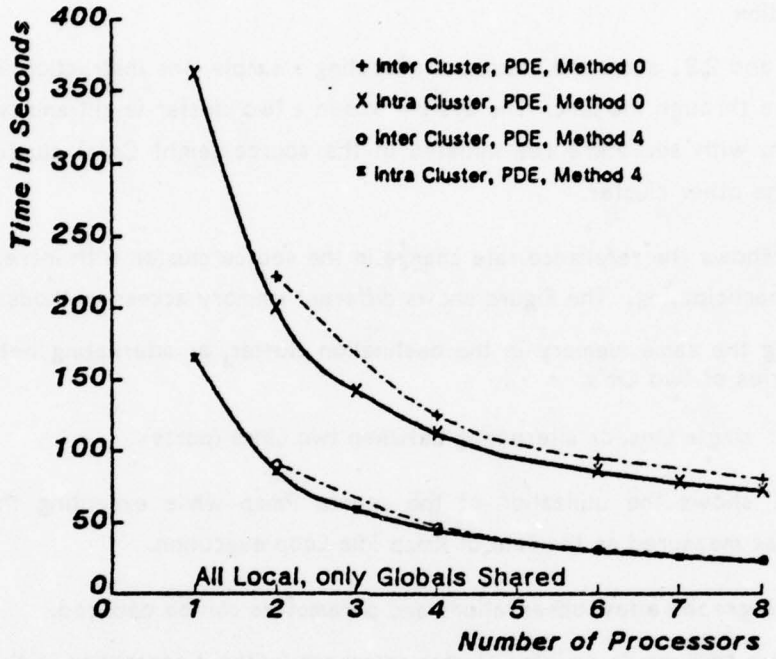


Fig. 2.18: Intra and Inter Cluster Execution Time

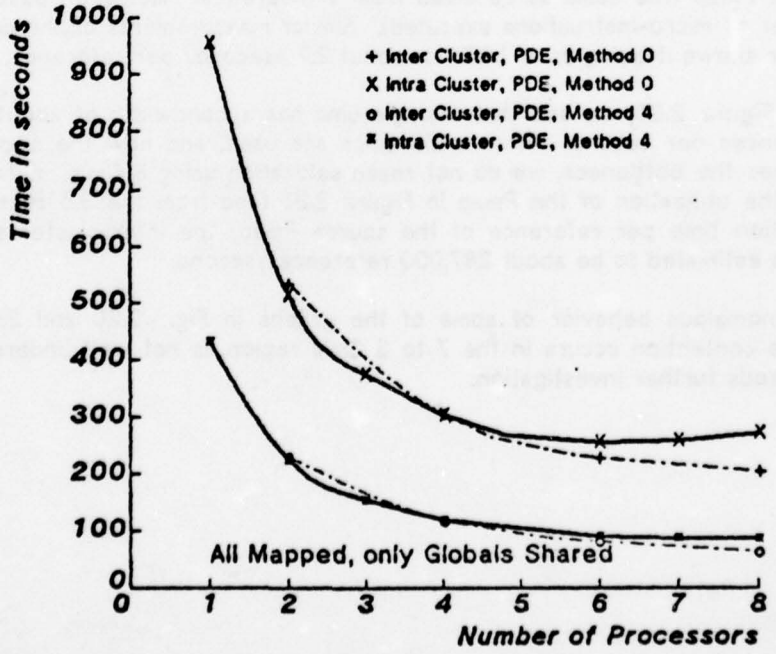


Fig. 2.19: Intra and Inter Cluster Execution Time

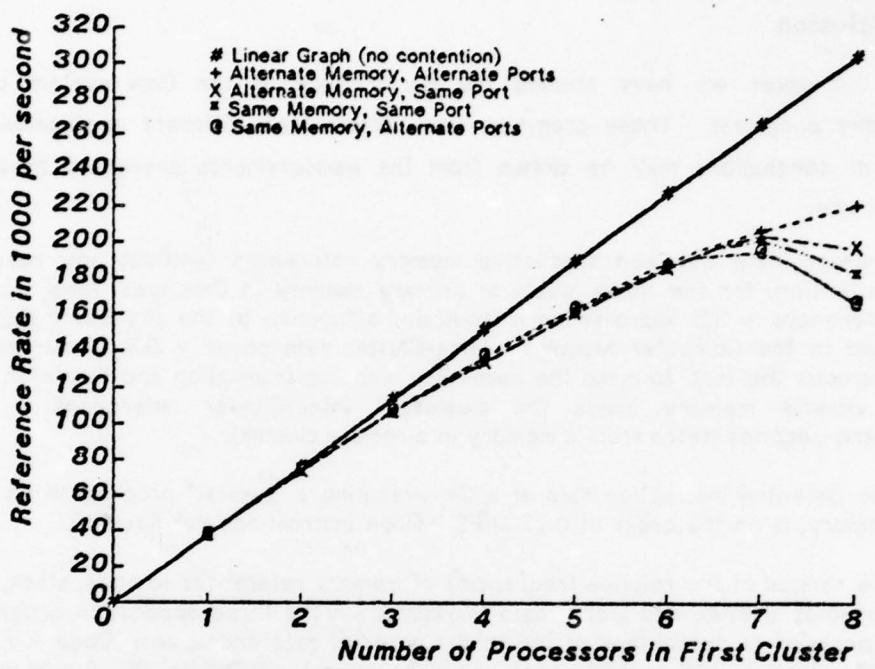


Fig. 2.20: Contention on Linc, Reference Rate

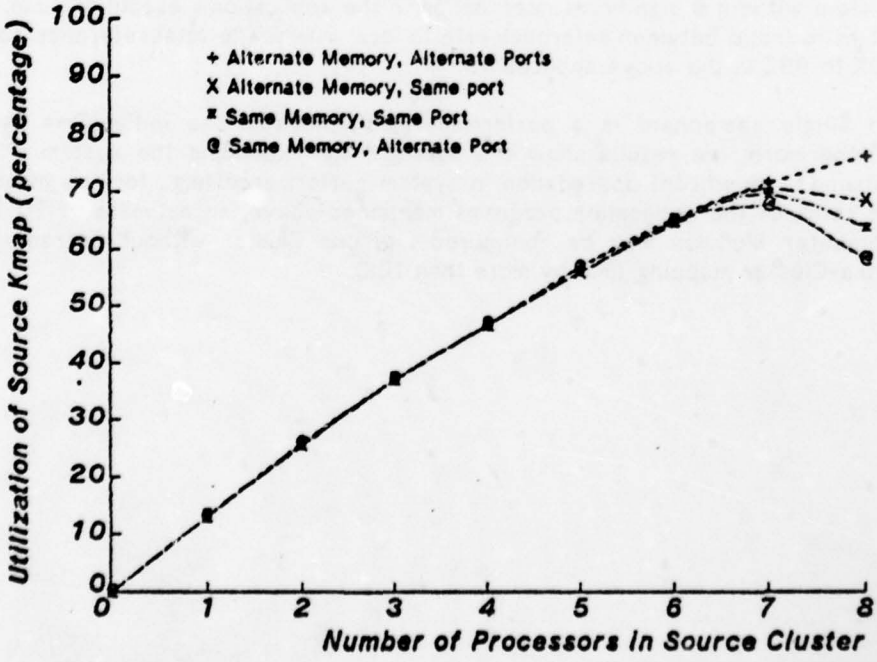


Fig. 2.21: Contention on Linc, Utilization of Source Kmap

2.6 Conclusion

In this chapter we have studied the performance of the Cm* system using three applications programs. Those programs were drawn from different application areas. A number of conclusions may be drawn from the measurements presented here; the most important are:

1. Average time between successive memory references (without any resource contention) for the three levels of primary memory in Cm* are: local memory references = 3.5 Micro-seconds (basically a function of the processor (LSI-11) used in the Computer Module). Intra-Cluster references = 9.3 Micro-seconds (includes the time to make the necessary mapping translation and the fetch from a remote memory inside the cluster). Inter-Cluster references = 26.4 Micro-seconds (fetch from a memory in a remote cluster).

2. The potential instruction rate of a Cm executing a "typical" program in its local memory, is on the order of 0.17 MIPS (Million Instructions Per Second).

3. The ranges of the relative frequencies of memory references to code, stack, local variables (owns), and global data (measured on the three application programs), expressed as percentage of the total number of references, are: Code = 71% to 82%. Stack = 10% to 25%. Local variables (owns) = 3.75% to 7%. Global data = 1.1% to 9.5%.

4. The difference in memory reference time, to the various memories in the memory hierarchy, makes the system's performance a function of the relative access frequency to these memories. The measurements show that by localizing code, stack and local variables (i.e., by replicating them in each participating processor's local memory), the global shared data can be put anywhere in the system without a significant degradation in the application's execution time. The hit ratio (ratio between reference rate to local memory to total reference rate) is 90% to 99% in the above applications.

5. No single component is a performance bottleneck in the initial Cm* system. Furthermore, the results show the potential for expanding the system without causing a significant degradation in system performance (e.g., for the measured hit ratio, of the application programs mentioned above, an estimate of 20 to 60 Computer Modules may be configured into one Cluster without degrading the Intra-Cluster mapping time by more than 10%).

3. Performance Models For Multiprocessors

3.1 Introduction

In this chapter we present several performance models and their application to evaluate the performance of Cm* like multiprocessor computer system.

The presentation of these models, starting from a simple, classical, one cluster M/M/1//N queue model to a relatively complex, multicluster, queueing network model reflects their real evolution in time and was driven by a need to reflect accurately the different aspects of the architecture and determine the effect of various possible parameter changes to it.

Three main factors most strongly influence the performance, or degradation of performance, of programs that utilize a multiprocessor architecture: the degradation due to lack of full parallelism in the program itself, the degradation due to the hardware structure and resource contention, and the degradation due to operating system interactions. The first factor is the subject of active research; we have tried to give some practical insight into this problem in Chapter 2 and in [Fuller 78]. The second factor (modelling the performance of the structure) is the aim of this chapter. It is important to model the structural performance for the system architect, as well as for many potential users that need to know and utilize the hardware structure without intervening levels of system software. We do not yet have enough data and experience to deal with the third factor seriously, though some initial investigations are being done by others in the Cm* project [Jones 78].

The methodology used in the derivation of each new model was first to validate its performance results to the actual system performance that was measured and reported in Chapter 2. We are fortunate to be able to compare theoretical and measured results. The model was then contrasted against the other models available and its advantages and limitations were explored - leading to the development of a new, better, model. The models were, then, applied to investigate many interesting performance issues.

In Section 3.2 the three one cluster Cm* models are described and compared. First, a short description of a one Cm* cluster, discrete time simulation is given. The basic M/M/1//N, one cluster, analytic queueing model is presented next. The effect of replacing the main exponential server by a deterministic server, M/D/1//N, is then explored. For validation, a comparison of the three models with the actual system is given in the end of that section. Appendices 1 and 2 contain the equations that were used to solve this M/D/1//N model. In Appendix 1 the detailed derivation of the Imbedded Markov chain solution, that we've derived for that model, are given. In Appendix 2 we show the main results obtained by Jaiswal [Jaiswal 1968], which used the Supplementary Variables approach to solve that model.

In Section 3.3 we describe and discuss the most complex, and useful, analytic model - a queueing network with classes of customers. There have been many publications on this subject in the last few years. Background and references to the main results are given. The evolution of this particular model from a simple one cluster model to a relatively complex multicluster model, is described. The computational algorithm, used to solve this model, is presented here (and we prove its correctness in Appendix 3). This model is then validated against the actual system and compared to the other models. Appendix 4 gives the algorithm used in the simple, one cluster queueing network model. Appendix 5 contains a list of programs that execute the various performance models.

In Section 3.4 the results of applying that model to explore many interesting performance issues are given, along with a discussion of the main conclusions. Section 3.5 summarizes this chapter.

3.2 One Cluster Models

3.2.1 Simulation Model

Analytic and simulation models are the two main performance modelling tools available to analyze computer performance. A discrete time, one cluster Cm*, simulation model was written by Brown [Brown 76]. This simulation model was modified and used as one of the models applied.[1]

Although this is quite a detailed model of the one cluster system, it suffers from the basic flaw of simulation that the results are not exact, and the simulation should run for a relatively long time in order to guarantee tight confidence levels for the results.

The time resolution of the simulation, and thus the accuracy, has a direct influence on its run time. e.g. changing the time resolution step from 100 nsec to 10 nsec increased the simulation time of a one cluster, 24 Cm's system from about 2 minutes to 20 minutes of KL10 CPU time - to acheive the same confidence levels on the results. Unlike simulation models, the precision of results is usually not an issue in analytic models. The accuracy of the model's representation of the actual system is usually the only concern.

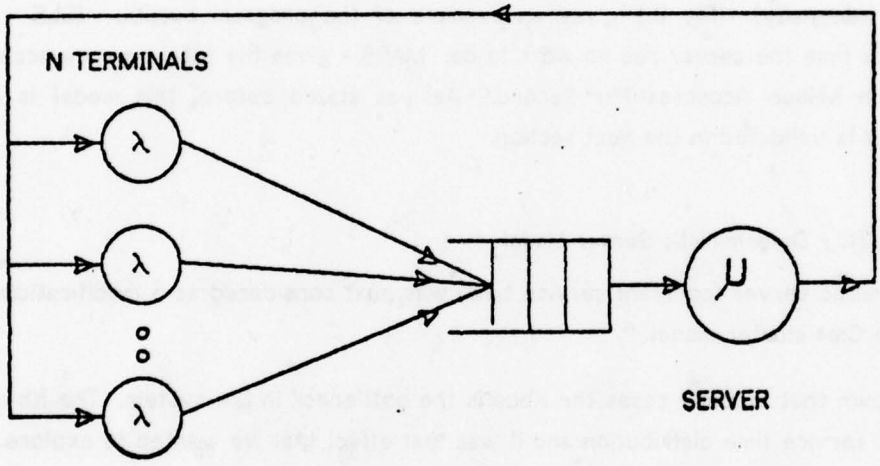
The simulation model has some very important attributes, like the possibilities to incorporate "blocking" in a node and inter-dependence between nodes (issues that will be discussed later) - which are not yet possible in a queueing network model. But the relatively

[1] This program is written in the Sail language.

long run time preclude its adequacy for the more complex multi-cluster structure and exact analytic models were next considered. See Fishman [Fishman 1973] for general discussion of discrete time simulation models.

3.2.2 M/M/1//N Queue Analytic Model

The first analytic model was the classical M/M/1//N Central Server "Machine Repairman Model" (Poisson arrival, Poisson server, one server, FCFS discipline, N customers). The machines are the Cm's in the Cm* single cluster and the "repairman" is the Kbus (which is known to be the bottleneck in the system in many cases).



The solution to this model is well known (e.g. [Kleinrock 75]) and will not be repeated here.

Using Kleinrock's notation:

λ - arrival rate from each terminal.

μ - service rate of the Poisson server.

N - number of terminals.

P_0 - probability of idle server.

$\bar{P}$ - average number of customers in the server.

Observing that from the balance equation:

Throughput = $(N - \bar{P})\,\lambda = (1 - P_0)\,\mu$

we can find $\bar{P}$ from: $\bar{P} = N - (1 - P_0)\,\mu/\lambda$

which require only the value of P_0 (and not the whole queue length distribution), where P_0 is received, as in Kleinrock, by:

$$P_0 = \left[\sum_{K=0}^{N} (\lambda/\mu)^K \cdot \frac{N!}{(N-K)!} \right]^{-1}$$

A program was written to solve the M/M/1//N equations. The mean service time demand from the server is the parameter that can be changed externally (some other parameters may be changed internally). Fig 3.1 gives an example of the program output. IDLE - is the percentage of time the server has no work to do. MAPS - gives the total memory accesses in the system in Million Accesses Per Second. As was stated before, this model is further discussed and is validated in the next section.

3.2.3 M/D/1//N - Deterministic Server Model

A deterministic server (constant service time) was next considered as a modification to the previous one Cm* cluster model.

It was known that in many cases the Kbus is the bottleneck in the system. The Kbus has a deterministic service time distribution and it was that effect that we wanted to explore.

Jaiswal [Jaiswal 68] showed the derivation of a M/G/1//N (general distribution) model using the Supplementary Variables technique. His main results and the derivation for the M/D/1//N are given in Appendix 2, and were used to evaluate this model by an APL program.

A more straightforward way to construct the M/D/1//N model was to use the Imbedded Markov chain approach (see [Kleinrock 75] for general discussion of this method). Because we could not find a solution to the M/G/1//N (or the M/D/1//N) models in the literature, we have derived it in Appendix 1.[1]

---

[1] Using the same APL program this solution was compared and was found to be identical to the Supplementary Variables solution, as expected

```
ENTER MEAN SERVICE TIME FOR K.MAP:
1.78
```

N / Miss Ratio	1 CM-S IDLE	1 CM-S MAPS	2 CM-S IDLE	2 CM-S MAPS	4 CM-S IDLE	4 CM-S MAPS	8 CM-S IDLE	8 CM-S MAPS	12 CM-S IDLE	12 CM-S MAPS	16 CM-S IDLE	16 CM-S MAPS	24 CM-S IDLE	24 CM-S MAPS
0.01	.995	.2810	.990	.5620	.980	1.1239	.960	2.2475	.940	3.3709	.920	4.4941	.880	6.7395
0.02	.990	.2764	.980	.5528	.961	1.1054	.921	2.2098	.882	3.3132	.843	4.4154	.764	6.6156
0.05	.977	.2636	.953	.5269	.906	1.0525	.813	2.0993	.721	3.1385	.629	4.1675	.450	6.1770
0.10	.956	.2446	.913	.4883	.827	.9725	.658	1.9236	.495	2.8395	.342	3.6954	.104	5.0354
0.15	.939	.2282	.879	.4547	.759	.9016	.530	1.7593	.323	2.5340	.157	3.1564	.011	3.7053
0.20	.924	.2139	.849	.4253	.701	.8387	.428	1.6076	.204	2.2363	.064	2.6302	.001	2.8065
0.25	.910	.2012	.822	.3992	.652	.7827	.346	1.4694	.126	1.9635	.025	2.1916	.000	2.2470
0.30	.899	.1900	.799	.3761	.609	.7329	.282	1.3450	.078	1.7262	.010	1.8542	.000	1.8726
0.35	.888	.1799	.779	.3554	.571	.6882	.231	1.2340	.049	1.5262	.004	1.5984	.000	1.6051
0.40	.878	.1709	.760	.3368	.539	.6482	.191	1.1357	.032	1.3600	.002	1.4018	.000	1.4045
0.45	.870	.1627	.744	.3200	.510	.6121	.160	1.0487	.021	1.2222	.001	1.2473	.000	1.2484
0.50	.862	.1553	.729	.3047	.484	.5794	.135	.9718	.014	1.1075	.000	1.1230	.000	1.1236
0.60	.848	.1423	.703	.2782	.442	.5229	.099	.8435	.007	.9296	.000	.9362	.000	.9363
0.70	.836	.1313	.681	.2558	.407	.4758	.075	.7421	.004	.7993	.000	.8025	.000	.8026
0.80	.826	.1219	.663	.2367	.379	.4361	.059	.6607	.002	.7005	.000	.7022	.000	.7022
0.90	.818	.1137	.647	.2202	.356	.4022	.048	.5944	.002	.6232	.000	.6242	.000	.6242
1.00	.810	.1066	.634	.2058	.336	.3738	.040	.5396	.001	.5612	.000	.5618	.000	.5618

MAPS = Million Accesses Per Second

FIG 3.1: OUTPUT RESULTS OF AN M/M/1//N MODEL

3.2.4 Comparing and Validating the Three, One Cm* Cluster, Models

Fig. 3.2 shows the the throughput results of the three models discussed so far, together with the measurement results of the actual system (taken from fig. 2.14, using the PDE benchmark) for the case when all the references are mapped through the Kmap back to the same Cm. As expected, the results of the simulation and the deterministic (M/D/1//N) models are very close to the actual measurements, while the results of the M/M/1//N model are slightly lower.

Price [Price 76] investigated the effect of the central server service time distribution on the performance of the models (this effect is shown again in Appendix 2). He showed that a deterministic service time gives the upper bound on the server utilization of all possible server distributions, and therefore throughput - as is shown on the graph.

From this validation one can conclude that the models give very accurate estimation of the actual system behavior for both the low and high traffic level cases (for the Kbus as a bottleneck in the system case). The M/M/1//N model give somewhat more pessimistic results than the actual system for the intermediate load case (about eight percent in the worst case), while the two other models are in the two percent error in the whole range.

There are many deficiencies to all these models, the most important of which are:

a. Inability to represent a multi-cluster structure by the analytic models and impracticality of using a simulation model.

b. For the analytic models, inability to evaluate the structure for the cases where there are more than one clear bottleneck in the system, or the bottleneck is not known a priori.

c. The analytic models are not detailed enough to enable the evaluation of different parameter changes (which are not represented in the model). For example, modelling the sharing of some of the references to one common memory module is not possible.

These limitations led to the investigation and development of the more promising queueing network models, that are discussed in the next section.

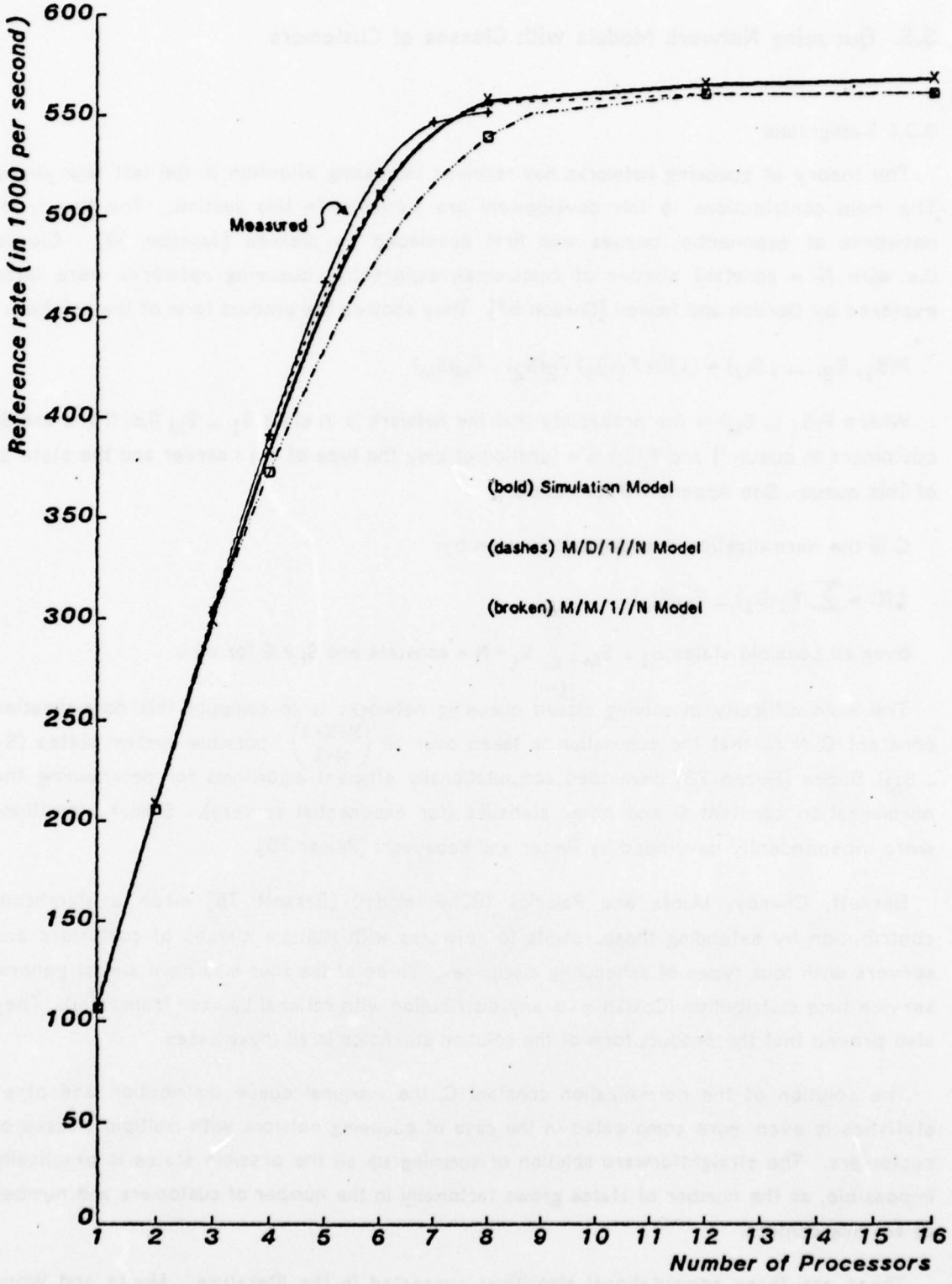


Fig. 3.2: One Cluster Models, Ref. Mapped Back to Same Cm

3.3 Queueing Network Models with Classes of Customers

3.3.1 Background

The theory of queueing networks has received increasing attention in the last few years. The main contributions to this development are surveyed in this section. The theory of networks of exponential queues was first developed by Jackson [Jackson 63]. Closed (i.e. with N = constant number of customers), exponential, queueing networks were later explored by Gordon and Newell [Gordon 67]. They showed the product form of the solution:

$$P(S_1, S_2, \ldots, S_M) = (1/C)\, F_1(S_1)\, F_2(S_2) \ldots F_M(S_M)$$

Where $P(S_1 \ldots S_M)$ is the probability that the network is in state $S_1 \ldots S_M$ (i.e. there are S_i customers in queue i) and $F_i(S_i)$ is a function of only the type of the i server and the state S_i of this queue. See Appendix 3 for details.

C is the normalization constant and is given by:

$$1/C = \sum F_1(S_1) \ldots F_M(S_M)$$

over all possible states $S_1 \ldots S_M \mid \sum_{i=0}^{M} S_i = N$ = constant and $S_i \geq 0$ for all i.

The main difficulty in solving closed queueing networks is to compute this normalization constant C. Note that the summation is taken over all $\binom{M+N-1}{M-1}$ possible system states ($S_1 \ldots S_M$). Buzen [Buzen 73] developed computationally efficient algorithms for determining the normalization constant C and other statistics (for exponential servers). Similar algorithms were independently developed by Reiser and Kobayashi [Reiser 73].

Baskett, Chandy, Muntz and Palacios (BCMP model) [Baskett 75] made a significant contribution by extending these results to networks with multiple classes of customers and servers with four types of scheduling disciplines. Three of the four may have almost general service time distribution (Coxian - i.e. any distribution with rational Laplace transform). They also proved that the product form of the solution still holds in all these cases.

The solution of the normalization constant C, the marginal queue distribution and other statistics is even more complicated in the case of queueing network with multiple classes of customers. The straightforward solution of summing up all the possible states is practically impossible, as the number of states grows factorially in the number of customers and number of service centers.

There are three computational algorithms suggested in the literature. Muntz and Wong

[Muntz 74] proposed a computational algorithm to compute these statistics that were used in the first, one Cm* cluster, queueing network models that we constructed and was found to be easy to apply and understand, but not easy to expand and not efficient for the more complex, multiple classes, queueing network model (see the algorithm details in Appendix 4).

Reiser and Kobayashi [Reiser 75], [Reiser 76] developed efficient algorithms for the solution of open, closed and mixed queueing networks (i.e. they allow external arrival and departures from the network). They used a generating functions and convolution theory to prove the correctness of the algorithms. These algorithms were also used in the QNET4 (analytic model program) and the RESQ programs at IBM ([Sauer 76] and [Sauer 77]) - that allow the user to specify and evaluate queueing networks in a unified way and solve them by analytic (QNET 4) or simulation techniques. Unfortunately, their derivation and proofs are very difficult to comprehend, follow or modify.

We decided to follow the generating functions approach given by Williams and Bhandiwad (WB model) [Williams 76]. This was also suggested in a survey by Gelenbe and Muntz [Gelenbe 76] in which they wrote about this technique "*it provides an ordinary and unified approach to the problem*".

The analysis in the WB model was not detailed or general enough to apply directly to the multi-cluster Cm* system model. In the next sub-section and Appendix 3 we have developed the required algorithms. They were found to be similar to those developed by Reiser [Reiser 76] but, hopefully, considerably easier to understand.

For general survey of queueing networks see [Baskett 75], [Gelenbe 76], [Kleinrock 75], [Reiser 75], [Reiser 76], [Candy 77]. Some validation results of applying this method to the investigation of performance issues begin to appear (see [Lipsky 77], [Bose 76], [Giammo 76]) but they all apply to multiprogrammed, single CPU - multiple I/O channels computer systems, and their level of detail and results were consequently not applicable to our study.

3.3.2 Evolution of Cm* Queueing Networks Models

In this sub-section the evolution of the queueing network models from a simple, one cluster, model to a relatively complex multi-cluster model is described. The main deficiencies of each model are stated. We have tried to keep the models as simple as possible and introduced more complexity only in cases where they seemed essential to obtain more accurate predictions. The first model is described in detail in order to stress the main points and decisions that were taken.

3.3.2.1 Simple, one cluster model

The first, simplified, model of the one cluster Cm* system is given below:

$P_{1,1;2,1} = 1$

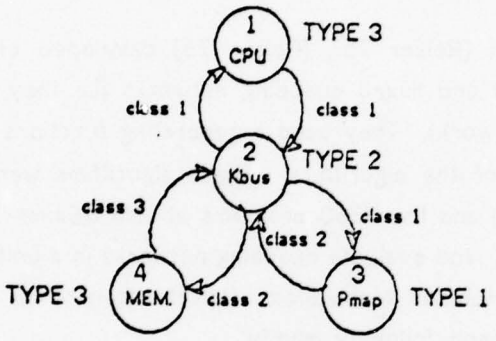
$P_{2,1;3,1} = 1$

$P_{3,1;2,2} = 1$

$P_{2,2;4,2} = 1$

$P_{4,2;2,3} = 1$

$P_{2,3;1,1} = 1$



Simple, One Cm* Cluster Model

Notation:

N - Number of customers (Cm's)

M - Number of service centers = 4

R - Number of classes of customers = 3

$P_{i,k;j,h}$ - Probability that a customer of class k in node i will next visit node j as class h.

$n_{i,r}$ - Number of customers of class r in node i.

By examining this simple model the main advantages of the queueing network technique become apparent. This includes the possibility to simulate movements of tokens (memory references in the Cm* case) from one node to the other, with a probability assigned to each possible move, the possibility of tokens to visit each node several times each time requesting a different service time from the server - thus allowing the use of this analytic modelling technique to model a detailed description of the system, which is almost as detailed as a detailed simulation of the system can be (the limitations of this analytic model compared to simulation model are discussed later).

From the four possible server types (see [Baskett 75] or Appendix 3), the types chosen for the nodes in the model are:

(a) service center 1 - We assumed an exponential (M) service time distribution between

external memory references, with mean inter-reference time of λ (the exponential assumption was not validated on the actual system). Type 3 - Infinite server (number of servers equal to the Number of Cm's) was chosen as the type of this node, i.e. one node represent all the CPU's in the system.

(b) Service center 2 - The requirement of a rational Laplace transform preclude the use of a deterministic server as the Kbus is ($F(S) = e^{-S/\mu}$). From the two possibilities Type 2 (Processor Sharing) or Type 4 (LCFS Preemptive Resume) - Type 2 is more adequate.

Note: There is a possibility to approximate a deterministic server by connecting several nodes in series (the "Methods of Stages"). This add to the complexity and was not used in the models.

(c) Service center 3 - Again a deterministic service station can't be used to represent the Pmap, and Type 1 - exponential server with FCFS discipline was used.

(d) Service center 4 - By ignoring contention in the memories (in this first model) a Type 3 (Infinite Server) was used to represent the memory delay, and the exponential service time distribution represent the spread of the DMA waiting time of the external reference.

Note: in all these models the mean DMA waiting time (which is competing for the memory cycle with the local Cm request) was modelled as a linear function dependent on the local Cm hit ratio. See the model listing for details.

The total number of states in this network model (given that there are L = 6 possible states $n_1, n_{2,1}, n_{2,2}, n_{2,3}, n_3, n_4$) is given by:

$$\text{Number of states} = \binom{L+N-1}{L-1} = \binom{5+N}{5}$$

N = 4 gives 126 possible states

N = 16 gives 20344 possible states

.
.
.

N = 50 gives 2118760 possible states

This shows the combinatorial way the number of states grows with the number of customers in the closed queueing network, which makes it difficult, even for such a simple model, to solve it directly by explicitly summing up all the possible states.

The algorithm to solve this model is a modification to the algorithm proposed by Muntz and Wong [Muntz 74] and is given in Appendix 4. The validation results are given in the end of

this section.

3.3.2.2 Model with sharing of memory module

Another node was added to the model to represent the effect of memory sharing interference (processors reference one shared memory module with probability α and "private" module - without interference - with probability $1 - \alpha$).

$P_{1,1;2,1} = 1$

$P_{2,1;3,1} = 1$

$P_{3,1;2,2} = 1$

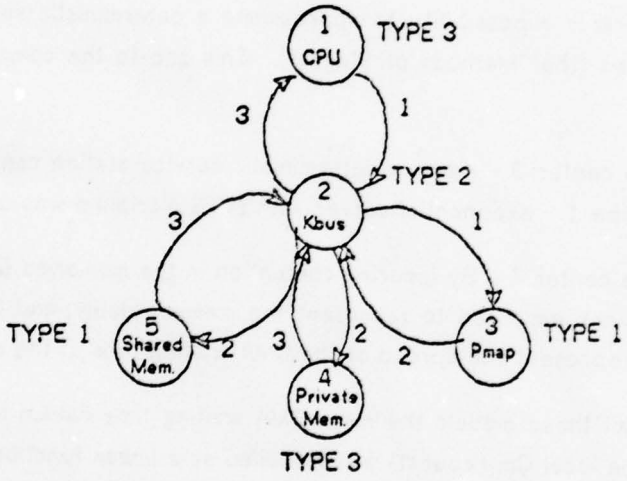
$P_{2,2;4,2} = 1 - \alpha$

$P_{2,2;5,2} = \alpha$

$P_{4,2;2,3} = 1$

$P_{5,2;2,3} = 1$

$P_{2,3;1,3} = 1$



First model with Memory Sharing

The solution of this model was similar to the solution of the previous model, with the addition of the new node. The simple change to the previous model shows the ease of making changes in the queueing network technique. Surprisingly, when we tried to validate the results of this model, they were found to be with higher values than the measured ones in the saturation region of the graph (in the case when all references where mapped to one shared memory module in the system). Investigation showed that this is a result of another difficulty of the queueing network technique - namely, that interdependence and blocking are not allowed in a node.

This is illustrated in the following case. Assume that a memory reference (token) is now in the Kbus (node 2) and would next move to the shared memory module. In the actual system, if another reference has already occupied this memory module the request will wait in the Kbus for the memory to become free and then will move to that module (with all the pertinent communication delays along this route). Only upon its return through the Kbus to the requested CPU can another waiting request start moving in the same route.

In the model, however, blocking of a reference as a result of a limited access to a subsystem - while servicing other requests by the Kbus, is not allowed.

In this shared model case, the reason for the inaccurate results lies in the fact that in this model tokens move immediately and queue at the shared memory module server (node 5) without blocking by the Kbus. The throughput of this node in saturation is higher than that of the system in saturation as the communication (Kbus and waiting delays) are done in parallel (in node 2) with the memory (node 5) servicing other requests - thus resulting in higher throughput.

Increasing the service time of the shared memory module to include the communications delay resulted in inaccurate results when the system was lightly loaded (accounting twice for the bus communication delays). As a compromise the model was modified so that shared references bypassed the Kbus and move directly from the Pmap (server 3) to the shared module (server 5), the service time of node 5 was changed to include the communication delays plus the DMA read time. The model used is then:

$P_{1,3;2,1} = 1$

$P_{2,1;3,1} = 1$

$P_{3,1;2,2} = 1 - \alpha$

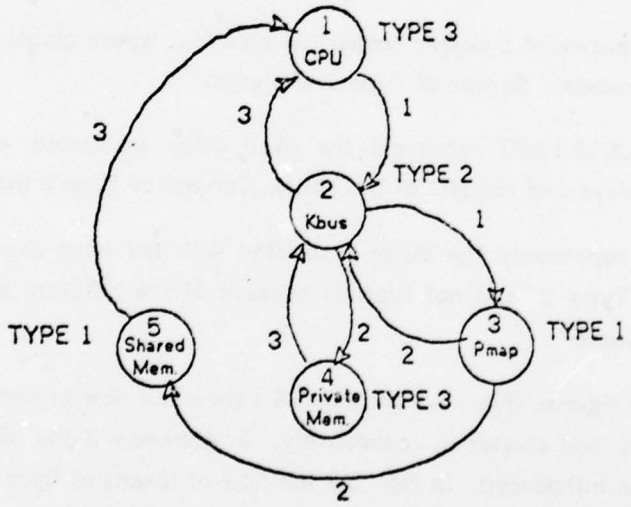
$P_{3,5;5,2} = \alpha$

$P_{2,2;4,2} = 1$

$P_{4,2;2,3} = 1$

$P_{5,2;1,3} = 1$

$P_{2,3;1,3} = 1$



Second, One Cluster, Shared Memory Model

An APL program solves this model using the algorithm discussed in Appendix 4. The

validation results are presented in the end of this section.

3.3.3 Multi-Cluster Cm* Network Model

3.3.3.1 The model

A model of a Cm* Multi-Cluster structure, with up to five clusters and variable number of Cm's, was constructed. The model is shown in Fig. 3.3 .

The five cluster network model is composed of 27 nodes.

Nodes 1,12,17,22,27 are CPU's (Type 3 servers).

Nodes 2,10,15,20,25 represent the Kbuses and are of Type 2 (P.S.) as a result of the different service rates for the different job classes.

Nodes 3,9,14,19,24 are the Pmap's servers of Type 3, as above.

Nodes 4,11,16,21,26 represent the private memory modules of each Cm. They are of Type 3 to represent the (almost) lack of contention when randomly accessing the private memories.

Node 5 represent a shared memory module (e.g. where global data base or synchronization primitives reside). Server of Type 1 was used.

Nodes 6,8,13,18,23 represent the fixed delay associated with an inter-cluster request (i.e. Linc delays and request of the Kbus). Servers of Type 3 (no queue) were used.

Node 7 represents the delay associated with the inter-cluster bus transaction. It is a server of Type 2 (and not type 1) because of the different service time requests for the different classes.

The two figures (Fig. 3.3 and Fig 3.4) show the flow of memory requests from the CPU's in cluster 1 and cluster 2, respectively. In Appendix 3 the notation of "global" and "local" classes was introduced. In Fig. 3.3 the flow of tokens of "global" class 1 - i.e. requests from server number 1 is seen. As an example we will follow the flow of these tokens (similar flows occur from the other cluster).

The *read* memory reference from node 1 leave this node (as "local" class 1) and request the Kbus. After Kbus delay they move (still as "local" class 1) to receive mapping service at the Pmap. From there they chose one of three possible routes:

(a) With probability α they move, as class 2, and queue at the shared memory module (5), following the memory access and bus delays they return to the CPU's (class 2).

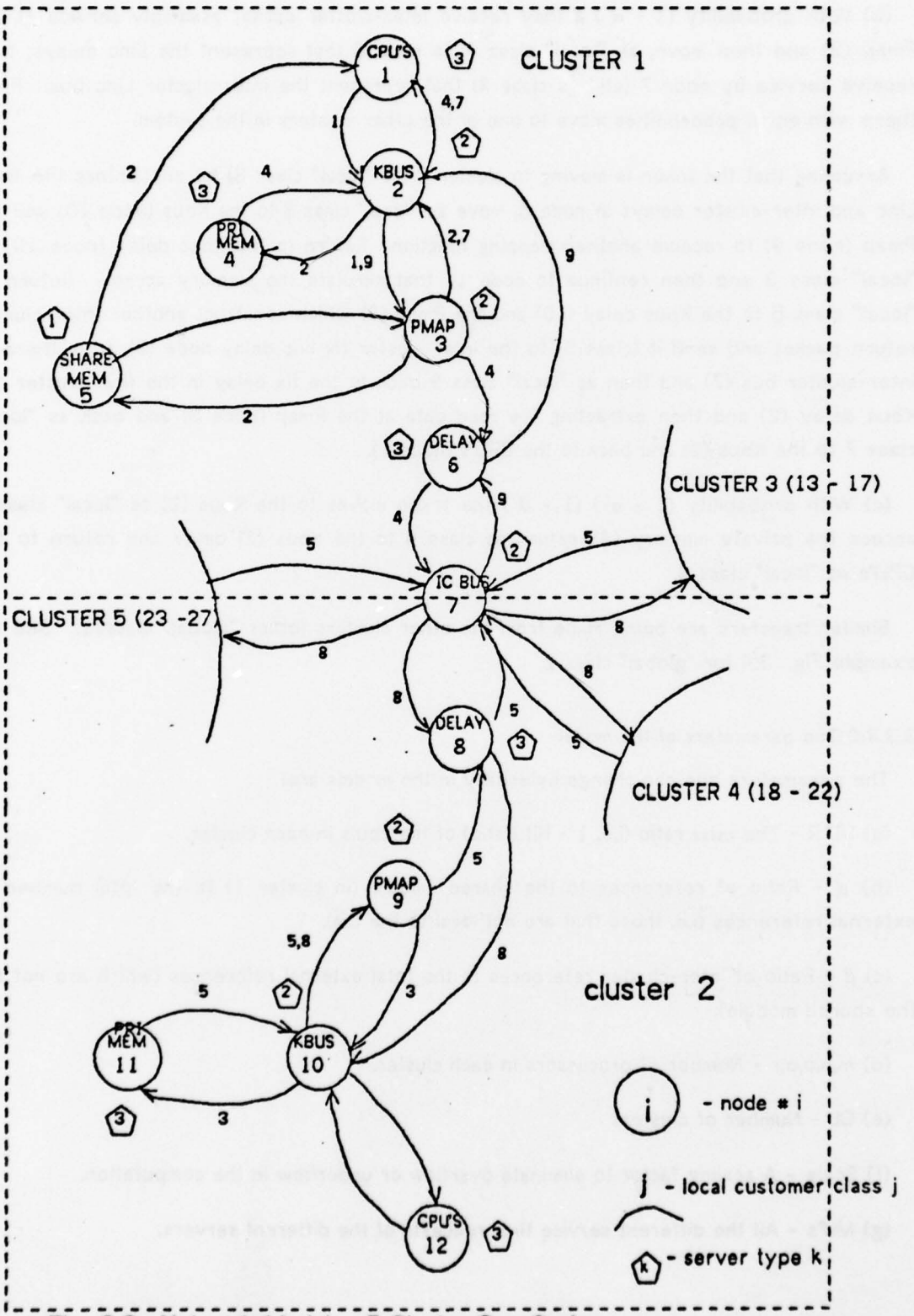


Fig. 3.3: Network Model - 5 Cluster Cm* System

clust1.spx/b

GLOBAL CLASS 1 (FLOW FROM FIRST CLUSTER)

(b) With probability $(1 - \alpha)\beta$ they receive inter-cluster packet assembly service at the Pmap (3) and then move, as "local" class 4, to node 6 that represent the Linc delays, then receive service by node 7 (still as class 4) that represent the inter-cluster Linc bus. From there with equal probabilities move to one of the other clusters in the system.

Assuming that the token is moving to cluster 2 (as "local" class 8) he encounters the fixed Linc and inter-cluster delays in node 8, move as "local" class 8 to the Kbus (node 10) and the Pmap (node 9) to receive another mapping function. Return to the Kbus delay (node 10) as "local" class 3 and then continue to node 11 that simulate the memory access. Return as "local" class 5 to the Kbus delay (10) and the Pmap (9) which construct another inter-cluster return packet and send it (class 5) to the inter-cluster fix linc delay node (8), from there to inter-cluster bus (7) and then as "local" class 9 back to the fix delay in the first cluster (6), Kbus delay (2) and then extracting the *read* data at the Pmap (node 3) and back as "local" class 7 to the Kbus (2) and back to the CPU's (node 1).

(c) With probability $(1 - \alpha)(1 - \beta)$ the token moves to the Kbus (2) as "local" class 2 access the private memory (4), return as class 4 to the Kbus (2) delay and return to the CPU's as "local" class 4.

Similar transfers are being made from the other clusters (other "global" classes). See for example Fig. 3.4 for "global" class 2.

3.3.3.2 The parameters of the model

The parameters one can change externally in the models are:

(a) HITR - The *miss* ratio (i.e. 1 - Hit Ratio) of the cpu's in each cluster.

(b) α - Ratio of references to the shared module (in cluster 1) to the total number of external references (i.e. those that are not local to the Cm).

(c) β - Ratio of inter-cluster references to the total external references (which are not for the shared module).

(d) m,n,o,p,r - Number of processors in each cluster.

(e) CL - Number of clusters

(f) Scale - A scaling factor to eliminate overflow or underflow in the computation.

(g) MU's - All the different service time requests of the different servers.

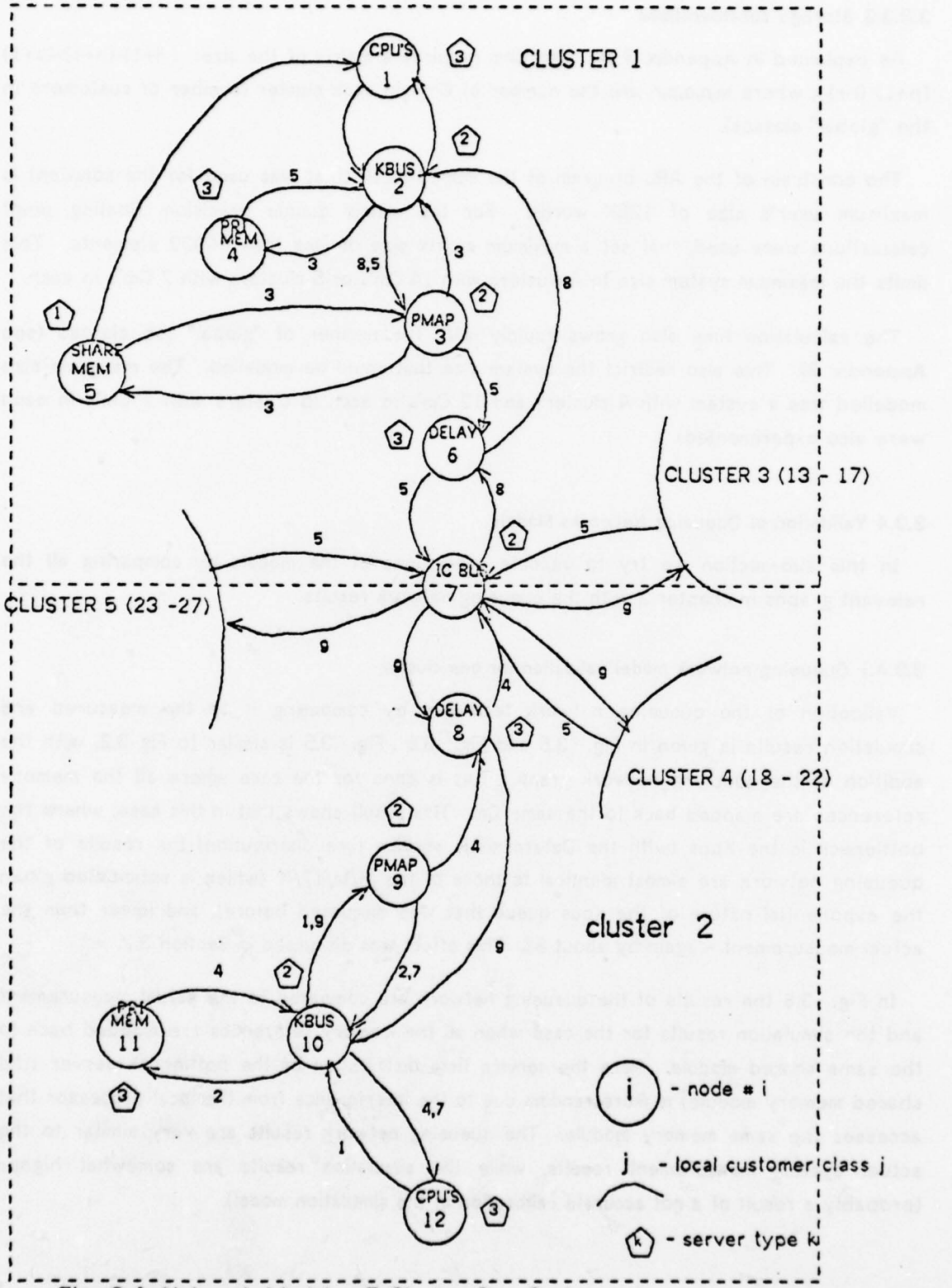


Fig. 3.4: Network Model - 5 Cluster Cm* System
global class 2 (flow from second cluster)

CLUST2.SPX/B

3.3.3.3 Storage considerations

As explained in Appendix 3 the algorithm requires a matrix of the size: $(m+1)\cdot(n+1)\cdot(o+1)\cdot(p+1)\cdot(r+1)$, where m,n,o,p,r, are the number of Cm's in each cluster (number of customers in the "global" classes).

The constrain of the APL program of the PDP10-KL10 (that was used for the solution) is maximum user's size of 128K words. For the matrix double precision, floating point calculations were used, that set a maximum matrix size of less than 64000 elements. This limits the maximum system size to 4 clusters with 14 Cm's or 5 clusters with 7 Cm's in each.

The calculation time also grows rapidly with the number of "global" job classes (see Appendix 3). This also restrict the system size that could be modelled. The maximum size modelled was a system with 4 clusters and 12 Cm's in each (5 clusters with 7 Cm's in each were also experimented).

3.3.4 Validation of Queueing Networks Models

In this sub-section we try to validate predictions of the models by comparing all the relevant graphs in chapter 2 with the queueing network results.

3.3.4.1 Queueing network model validation for one cluster

Validation of the queueing network technique by comparing it to the measured and simulation results is given in Fig. 3.5 and Fig. 3.6 . Fig. 3.5 is similar to Fig 3.2, with the addition of the queueing network graph. This is done for the case where all the memory references are mapped back to the same Cm. This result shows that in this case, where the bottleneck is the Kbus (with the Deterministic service time distribution) the results of the queueing network are almost identical to those of the M/M/1//N, (which is anticipated given the exponential nature of the Kbus queue that was discussed before), and lower than the actual measurement - again by about 8%. This effect was discussed in Section 3.2.

In Fig. 3.6 the results of the queueing network are compared to the actual measurement and the simulation results for the case when all the memory references are mapped back to the same shared module. Here the service time distribution of the bottleneck server (the shared memory module) is more random due to the interference from the local processor that accesses the same memory module. The queueing network results are very similar to the actual system measurement results, while the simulation results are somewhat higher (probably a result of a not accurate calibration of the simulation model).

3.3.4.2 Queueing network model validation for multi-clusters

Fig. 3.7 compares the queueing network to the measured data for the case of two Cm* clusters when all the references are from one cluster to memory module in the other cluster. The error is only in the 3-4% range.

In Fig. 3.8 we have tried to verify an experiment that was reported in Chapter 2 (Fig. 2.19) in which it was shown that for the extreme case when all the references are mapped, the saturation in the Kbus is dominant, and two clusters with equal number of Cm's in each (and 2.5% of the references mapped to one global module in one of the clusters) performed better than a one cluster with the same number of Cm's as the sum in the two clusters.

In the figure it can be seen that starting with 3 - 4 Cm's, the total number of references, in the two clusters case, is clearly higher than that of the one cluster, thus verifying the measured results.

The conclusion of all these validation data is that a single queueing network model with classes of customers captured the main factors in the actual structure, as was verified by all the different validation cases that we were able to test.

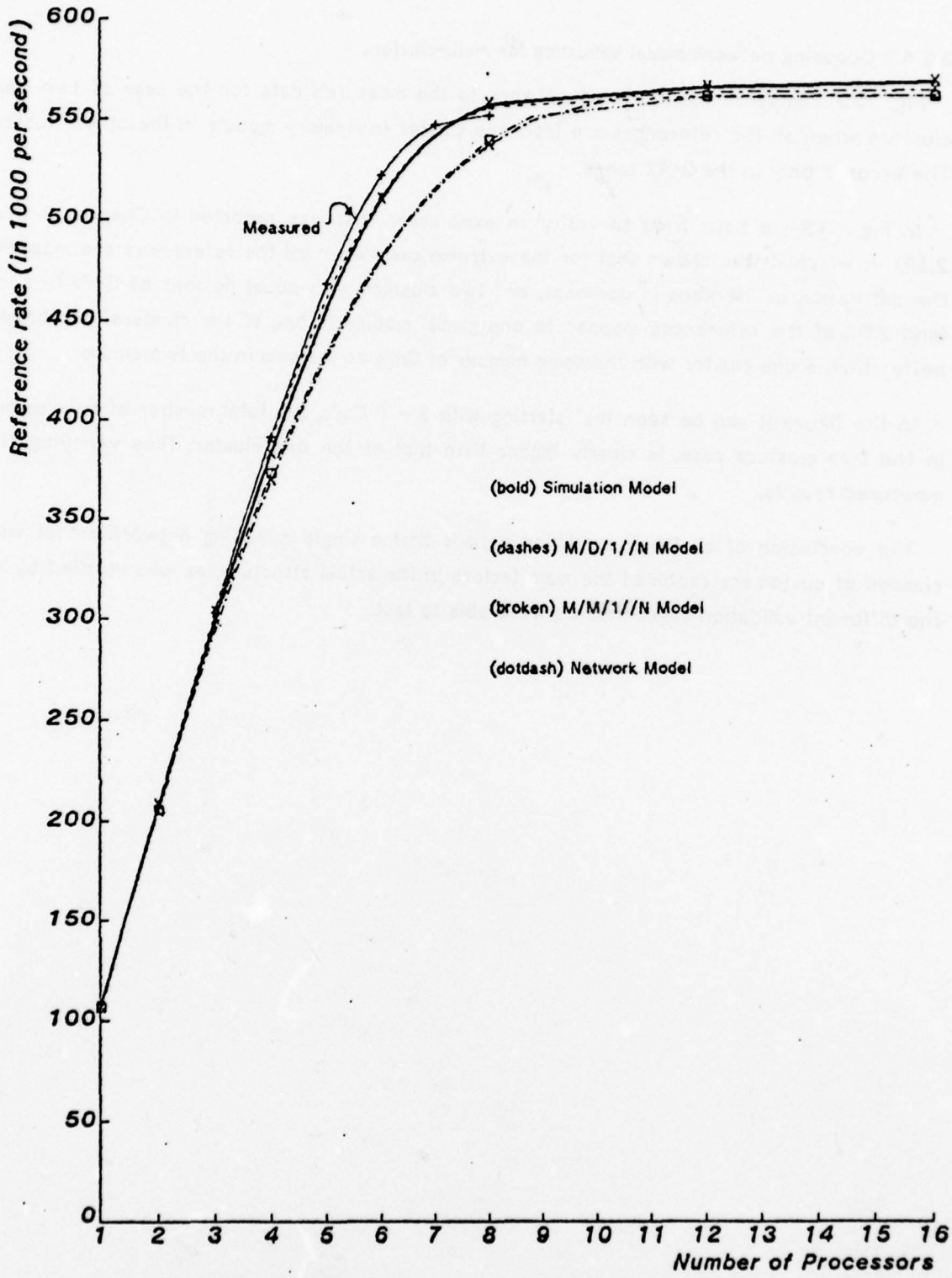


Fig. 3.5: All One Cluster Models, Ref. Mapped Back to Same Cm

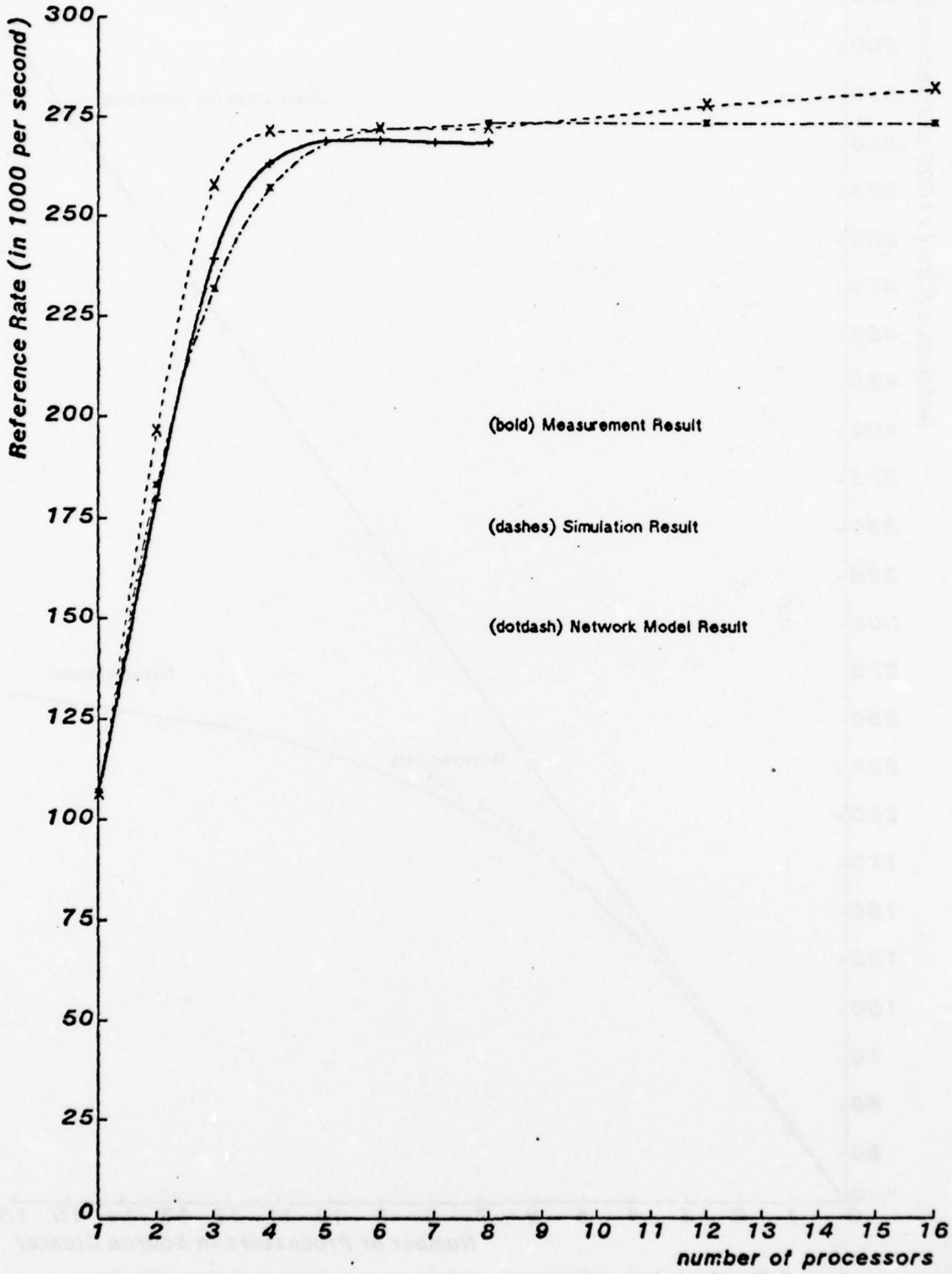


Fig. 3.6: One Cluster Models, All Ref. Mapped to Shared Memory Modul

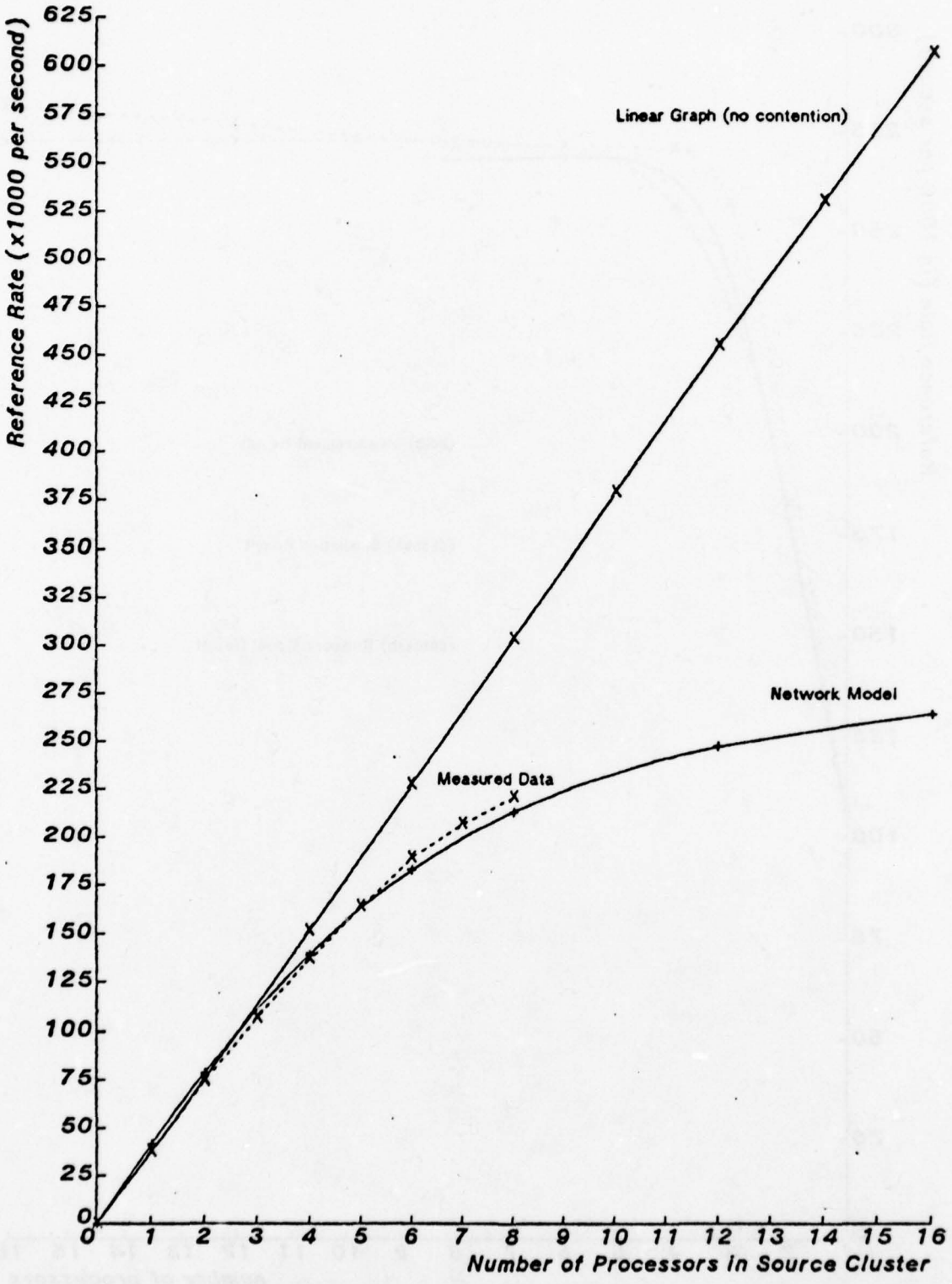


Fig. 3.7: Model Verification, All References are Inter-Cluster

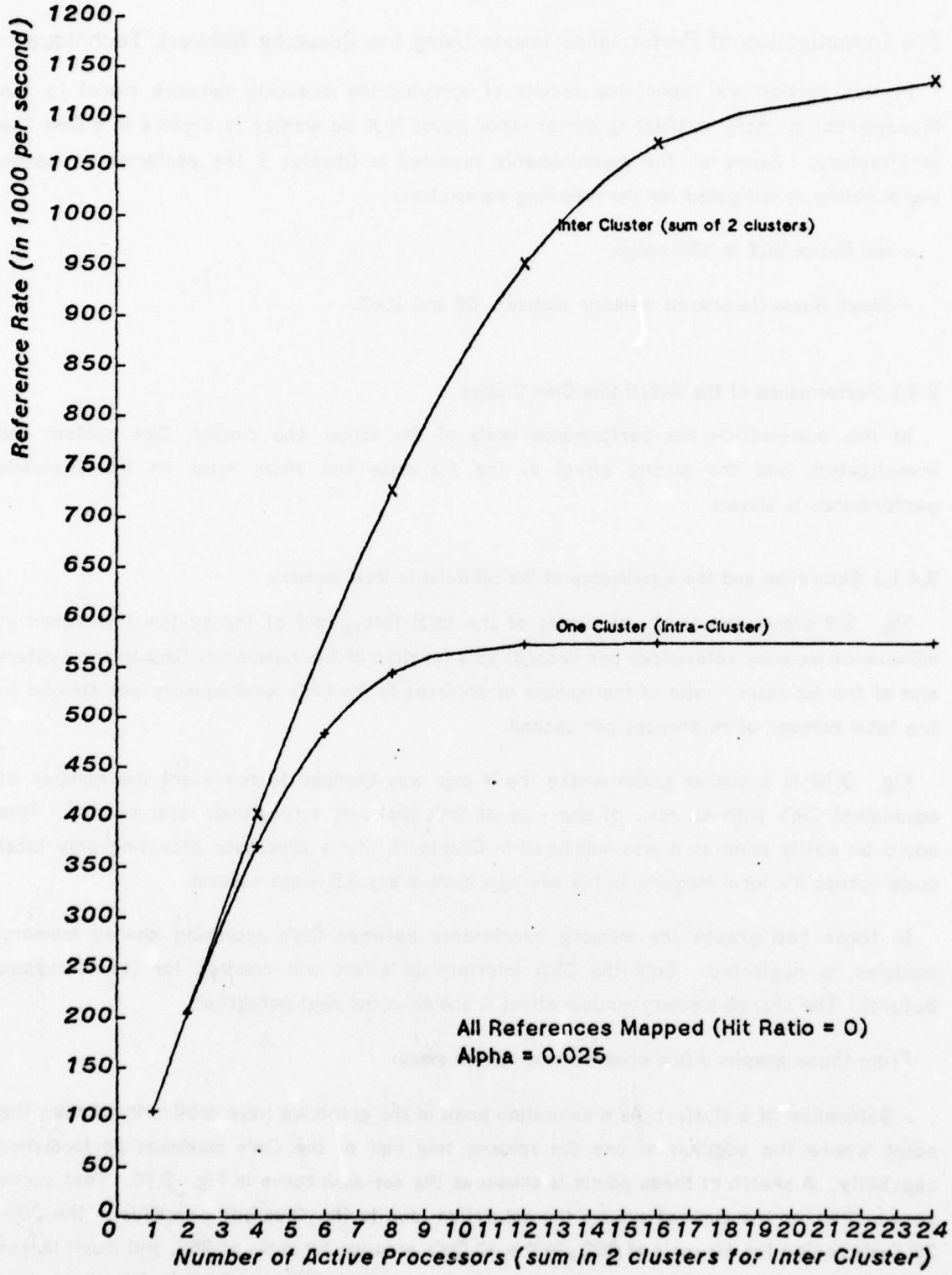


Fig. 3.8: Model Verification, Comparing One & Two Clusters

3.4 Investigation of Performance Issues Using the Queueing Network Technique

In this section we report the results of applying the queueing network model to the investigation of many interesting performance issues that we wanted to explore in a Cm* like architecture. Based on the measurements reported in Chapter 2 the performance issues were mainly investigated for the following parameters:

- *Hit Ratio*: 90% to 95% range.

- *Share Ratio* (to shared memory module): 0% and 100%.

3.4.1 Performance of the Actual One Cm* Cluster

In this sub-section the performance limits of the actual, one cluster, Cm* system are investigated, and the strong effect of the *hit ratio* and *share ratio* on the system's performance is shown.

3.4.1.1 Saturation and the significance of the Hit Ratio to local memory

Fig. 3.9 shows the family of curves of the total throughput of the system (measured in millions of memory references per second) as a function of the number of Cm's in the system and of the *hit ratio* - ratio of the number of accesses to the Cm's local memory per second to the total number of references per second.

Fig. 3.10 is a similar graph where the Y axis was changed to represent the number of equivalent Cm's with *hit ratio* of one - i.e. of Cm's that only access their local memory. This could be easily done as it was measured in Chapter 2 that a processor accessing only local code access it's local memory, in the average, once every 3.5 micro-second.

In these two graphs the memory interference between Cm's accessing shared memory modules, is neglected. Only the DMA interference effect was counted for (as discussed before). The shared memory module effect is shown in the next paragraph.

From these graphs a few observations can be made:

- Saturation of a cluster: As a saturation point in the graph we have arbitrarily chosen the point where the addition of one Cm returns only half of the Cm's maximum performance capability. A sketch of these points is shown as the *dot-dash* curve in Fig. 3.10. That curve shows that for the current system the saturation (due to the Kbus bottleneck) is in the 20 - 25 Cm's region for *hit ratio* of 90% , in the 40 Cm's area for *hit ratio* of 95%, and much larger for higher *hit ratios*. That shows that for a typical application a one Cm* cluster could have

been built with more Cm's without much performance loss

- The *hit ratio* has a very strong effect on the system's performance, e.g. 5 percent change in the *hit ratio* from 90% to 95% roughly doubles the performance in the saturation region. That emphasizes the importance of localizing the program in the Cm's local memory.

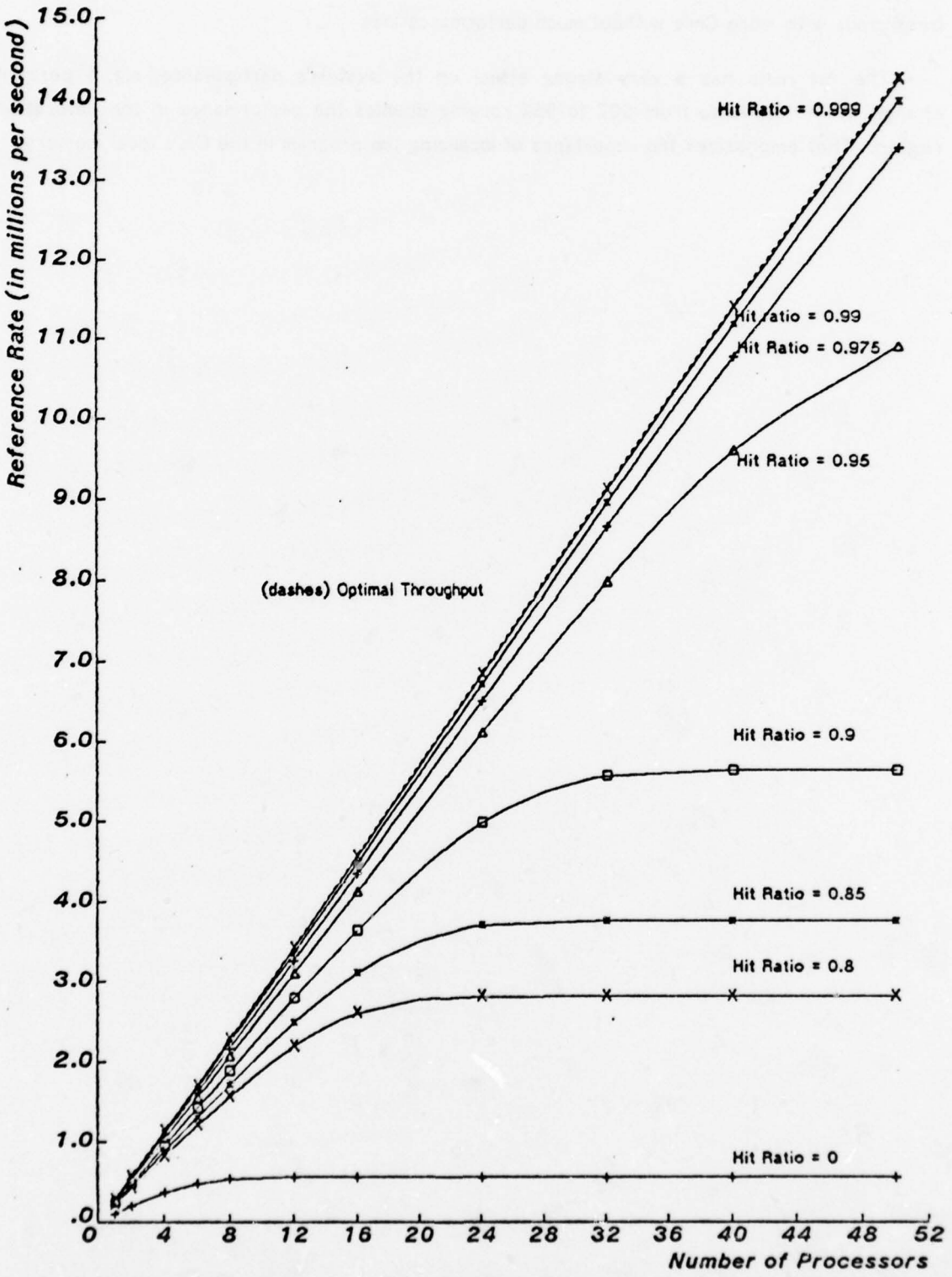


Fig. 3.9: Network Model, Throughput with Various Hit Ratios (1 Cluster)

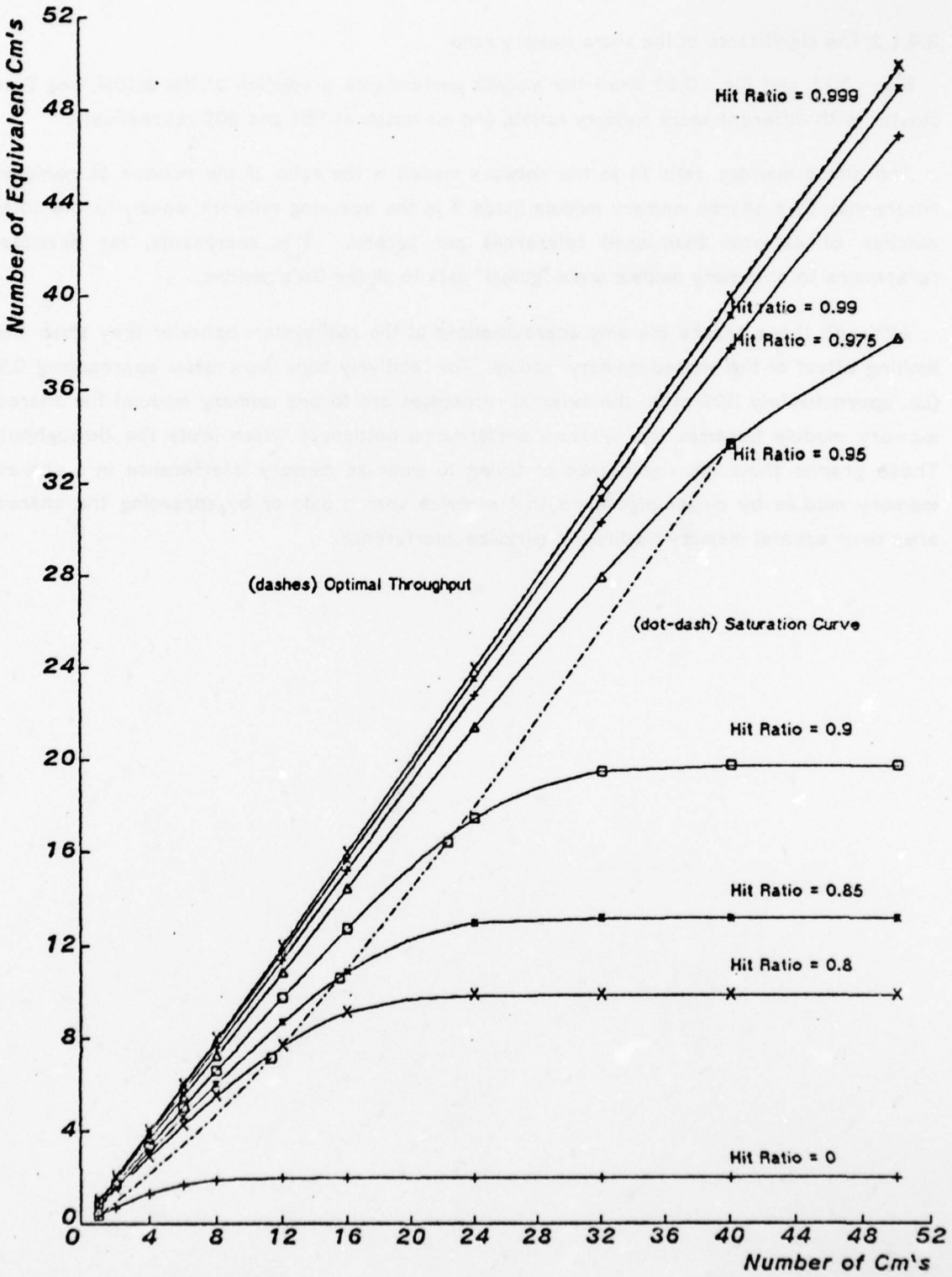


Fig.3.10: Network Model, Throughput with Various Hit Ratios(1 Cluster)

3.4.1.2 The significance of the share memory ratio

FIg. 3.11 and Fig. 3.12 show the model's performance prediction of the actual, one Cm* cluster with different *share memory ratio's*, and *hit ratio's* of 95% and 90% respectively.

The share memory ratio (a in the network model) is the ratio of the number of nonlocal references to a shared memory module (node 5 in the queueing network model) to the total number of external (non-local) references per second. This represents, for example, references to a memory module were "global" data to all the Cm's resides.

Although these graphs are only approximations of the real system behavior they show the limiting effect of the shared memory module. For relatively high *share ratios* approaching 0.5 (i.e. approximately 50% of all the external references are to one memory module) the shared memory module becomes the system's performance bottleneck which limits the throughput. These graphs show the significance of trying to minimize memory interference in a shared memory module by clever algorithms that minimize shared data or by spreading the shared area over several memory modules to minimize interference.

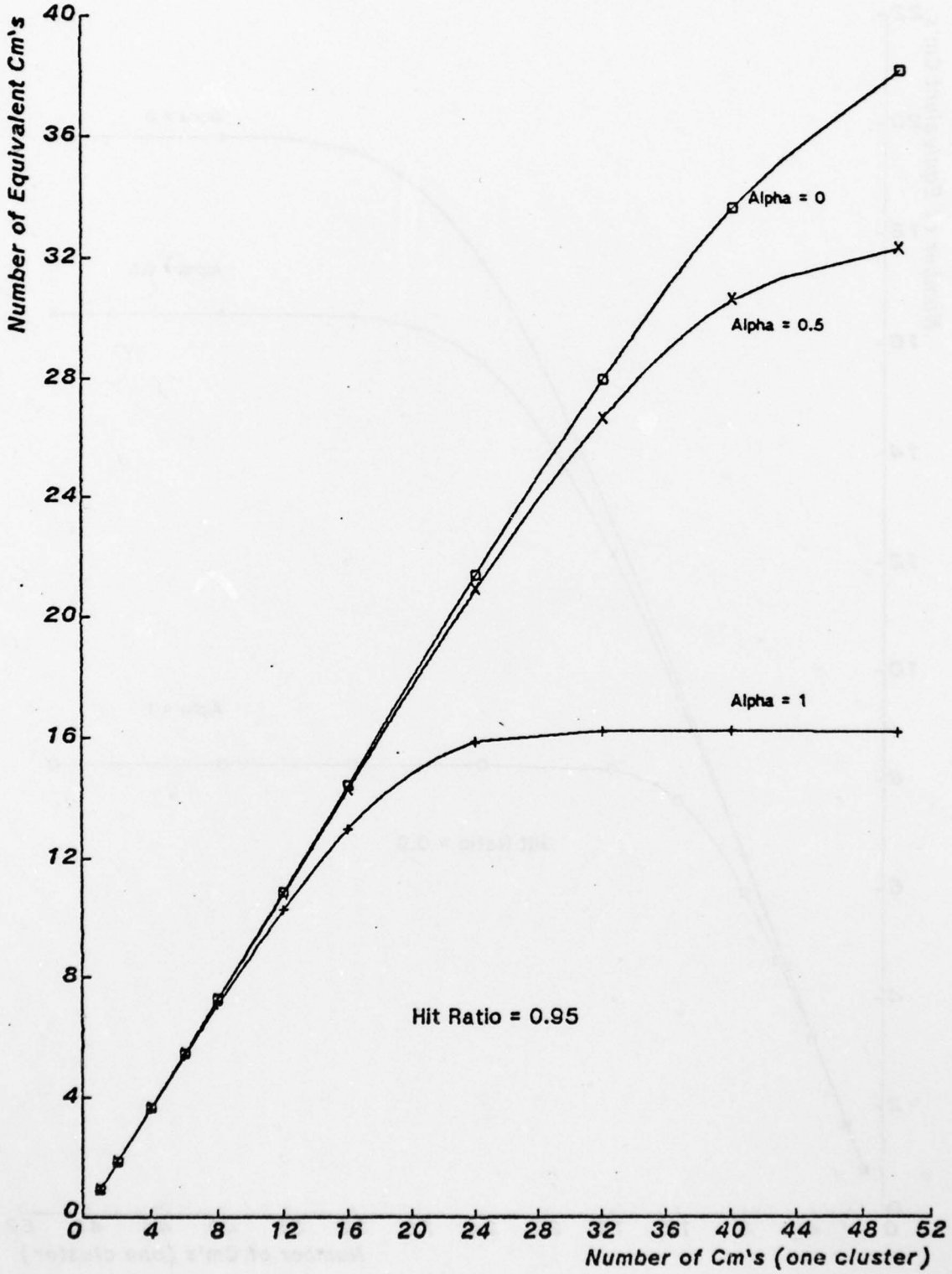


Fig. 3.11: Network Model, One Cluster, Effect of Share Ratio

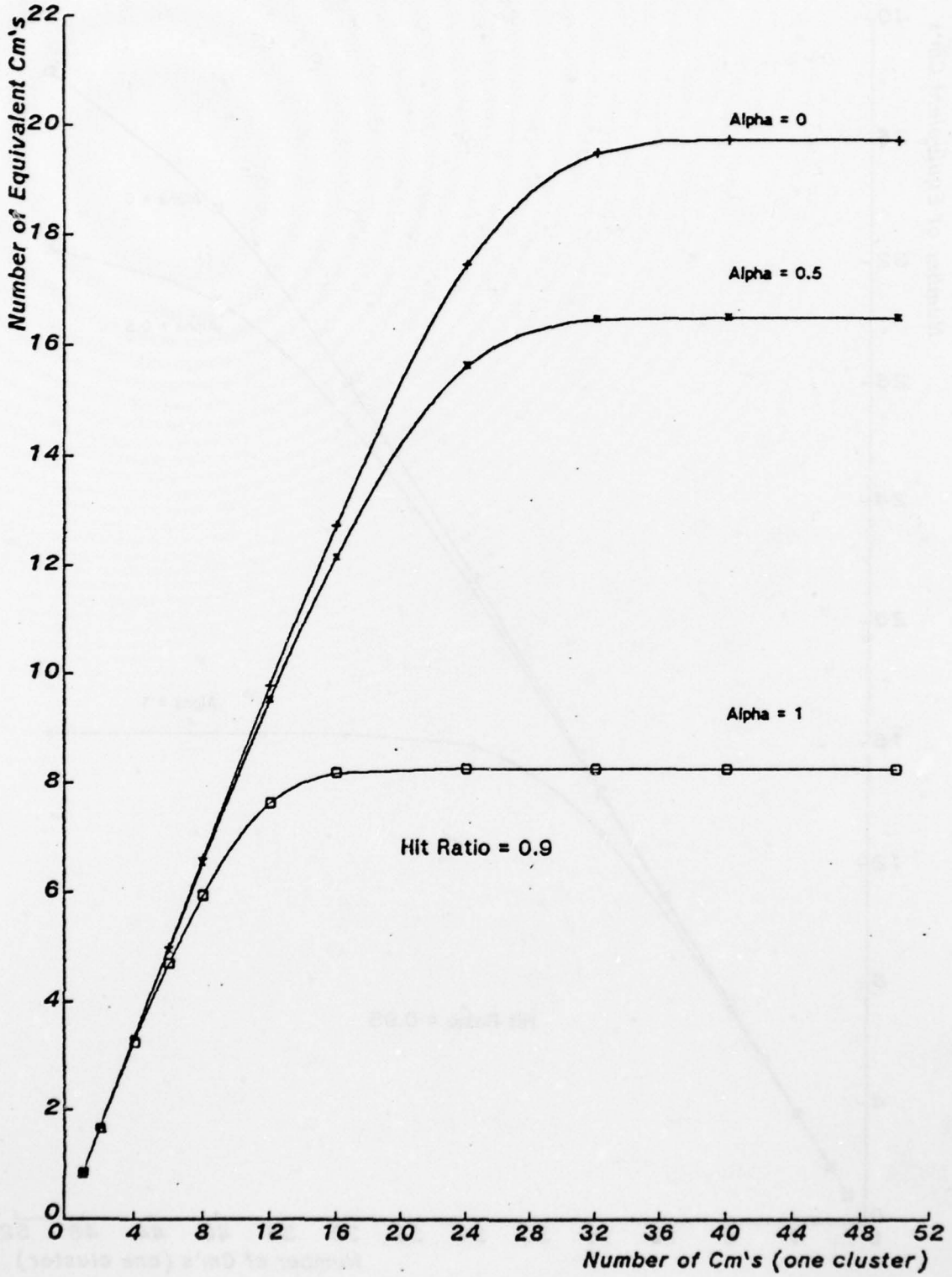


Fig. 3.12: Network Model, One Cluster, Effect of Share Ratio

3.4.2 Performance of One Cm* Cluster - Changing System Parameters

In this sub-section the impact of various system parameters changes on the performance is investigated. Some of these changes are the result of possible potential modification to the system and the others try to explore the architecture limitations and sensitivity to changes.

3.4.2.1 Changing the number of contexts

In [Swan 77b] the notion of the Kmap *context* was introduced and described in some detail. One *context*, out of a possible eight, is assigned to each *mapped* (i.e. non *local*) operation by the Pmap. Each *context* has a set of Pmap general purpose and subroutine linkage registers, in which the necessary computation for this operation is done. A *context* is released only upon completion of the operation. We wanted to explore the influence on performance of changing N - the maximum number of *contexts* available (which is part of the Pmap implementation cost).

By trying to do so we encountered again the same queueing network deficiencies - inability to *block* access to a subsystem and not allowing inter-dependence between nodes. These deficiencies inhibit the incorporation of *contexts* in the Kmap. Note that we have to block the access to the Kmap if N *contexts* are currently been used, and a *context* busy state has to be remembered and not assigned again while the reference is moving to another node in the system without releasing the *context*. We employed two techniques to overcome the problem:

(a) Instead of *blocking* the access to the Kmap we allow, as a crude approximation, any number of customers to access the Kmap simultaneously - but recorded in the experiments the probability that more than N contexts are busy. This can be done as we are able to calculate, in the queueing network technique, the queue length distribution in various nodes.

Figures 3.13, 3.14 and 3.15 show the probabilities results for the current 8 *contexts*, for 7 *contexts* (which was chosen as a possibility of reserving one *context* in the current implementation for some special error condition) and for four contexts. As can be seen from the figures, eight and seven contexts give very low probabilities of missing a context for the maximum cluster size of 14 Cm's, even for *hit ratio* in the 85% range (which explains why we had not experimented with more than eight contexts). Four *contexts* seems too few (more than 10% probability of a missing *context* for 90% *hit ratio* and more than 30% for the 85% *hit ratio*).

(b) To obtain more accurate results we had to resort to the simulation model and check the influence of *context* sizes on the system's throughput. Figures 3.16, 3.17 and 3.18 show the

system's throughput results with one to "infinite" number of *contexts* and *hit ratio's* of 80%, 90% and 95%, respectively. The results show that more than eight *contexts* would not add to the system performance (even for a large, one cluster, Cm* system), while with four contexts we might suffer 15% to 20% degradation in the system's throughput for *hit ratio* of 80% and large (more than 16 Cms) one cluster system.

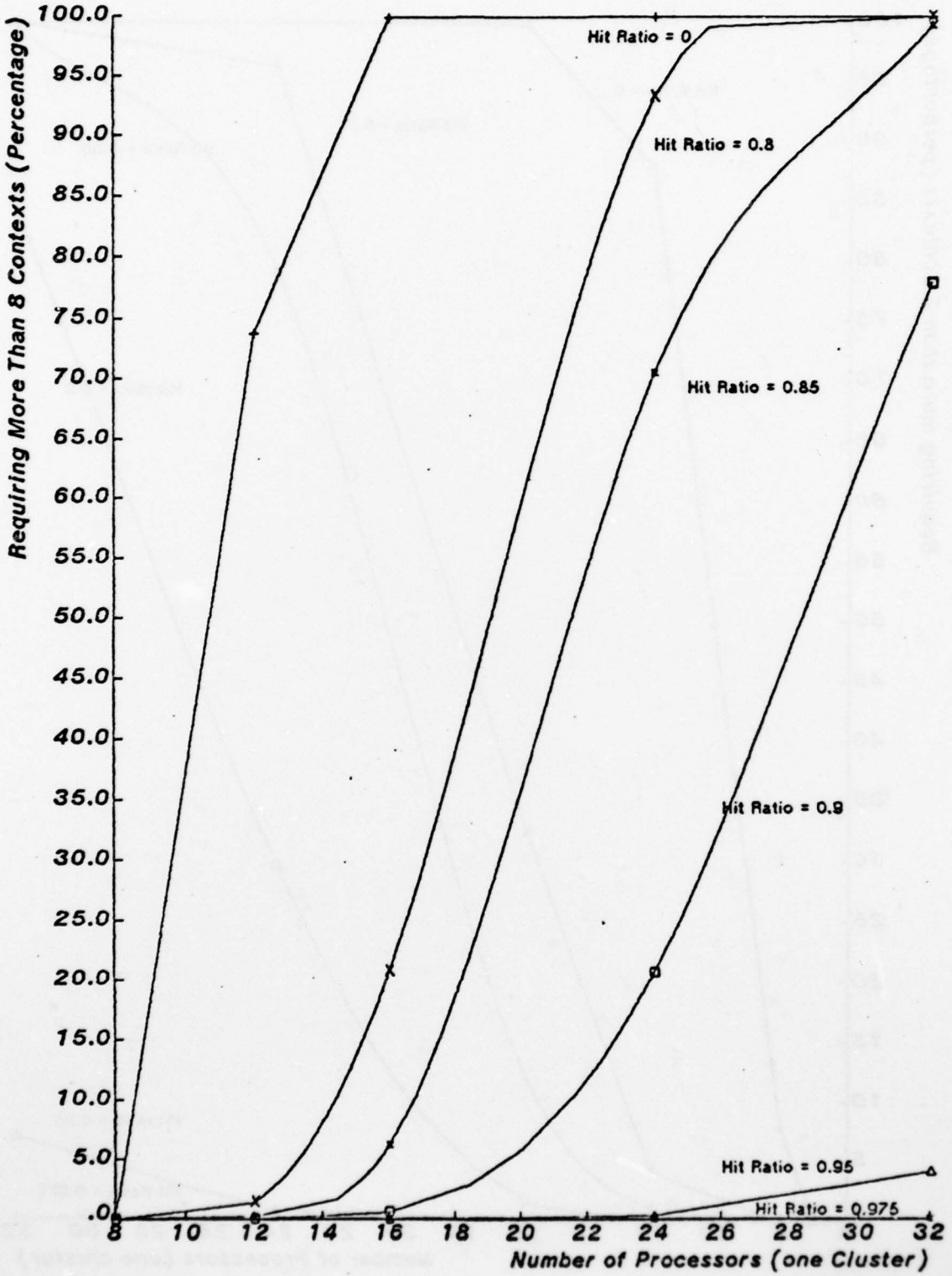


Fig. 3.13: Network Model, Requirement for more than 8 Contexts

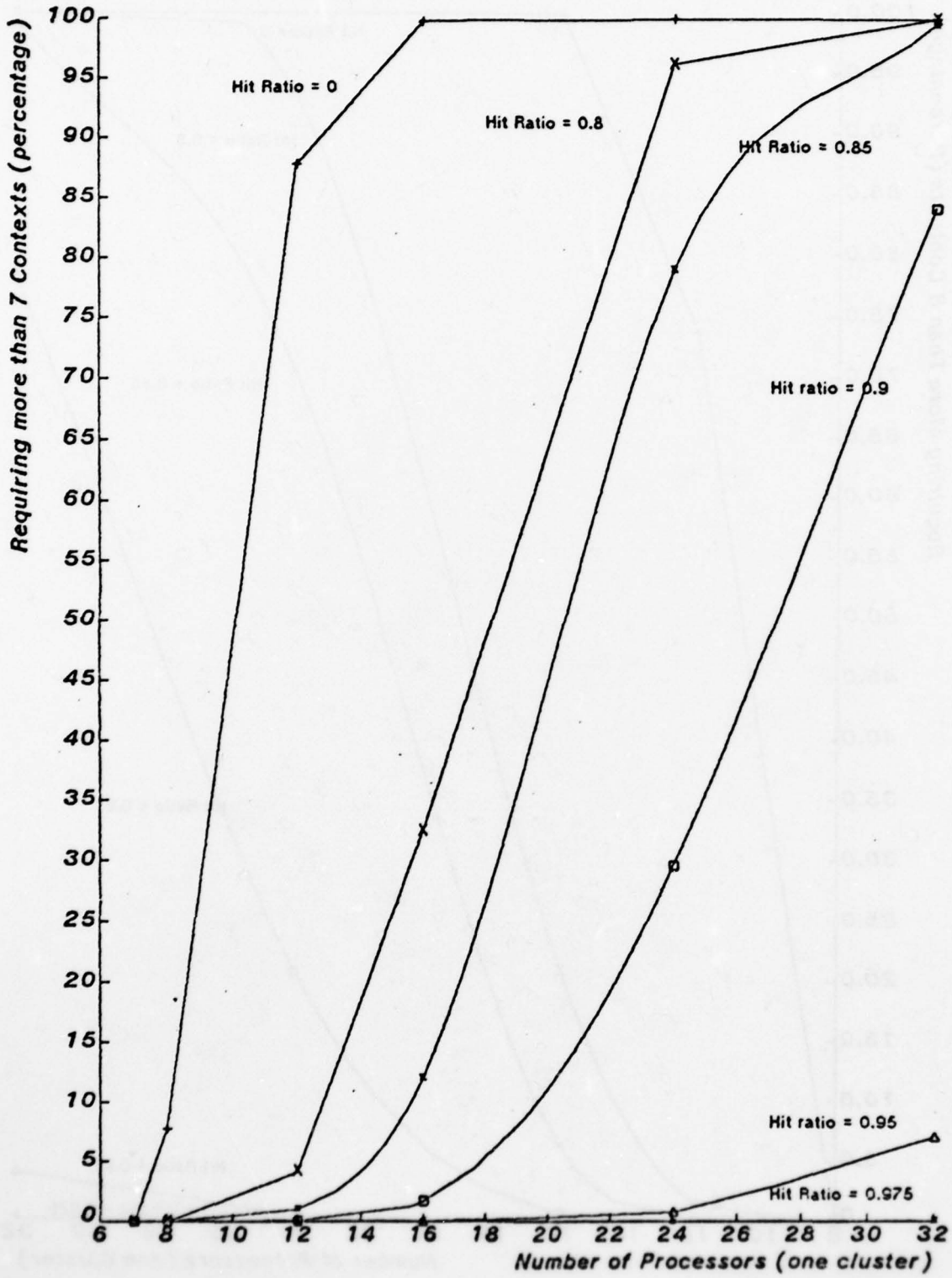


Fig. 3.14: Network Model, Requirement for more than 7 Contexts

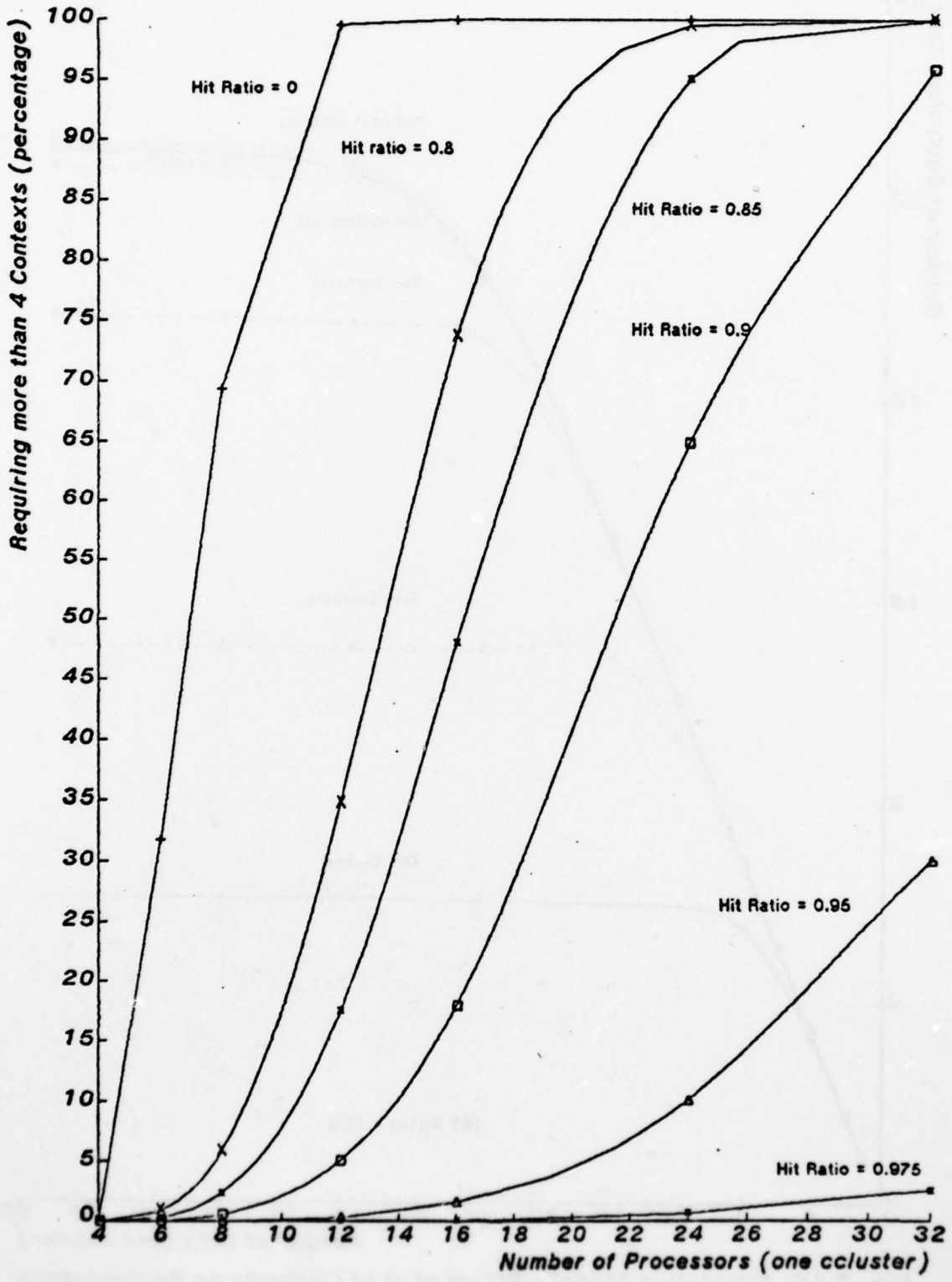


Fig. 3.15: Network Model, Requirement for more than 4 Contexts

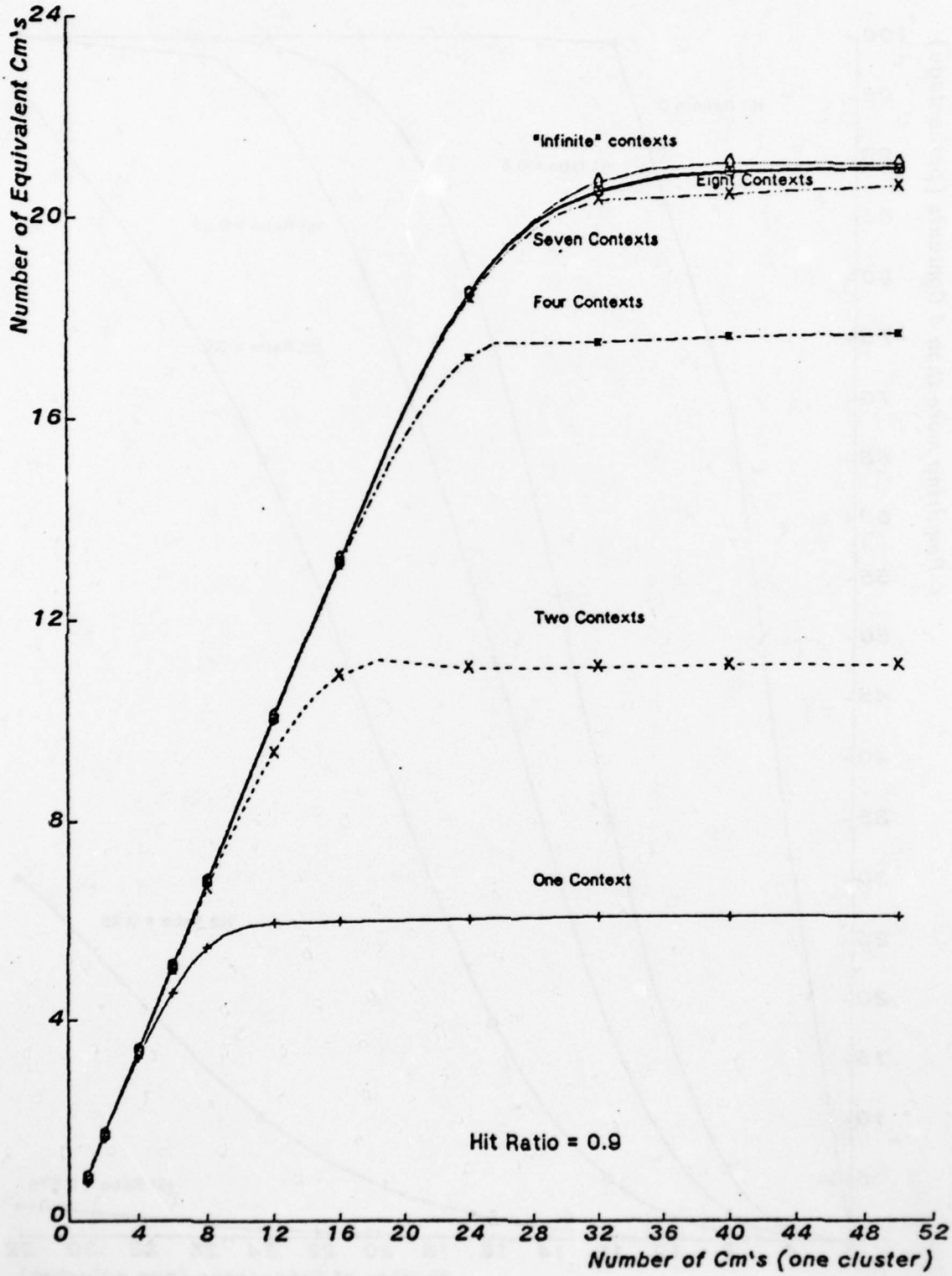


Fig. 3.16: Simulation Model - Effect of # of Contexts on Performance

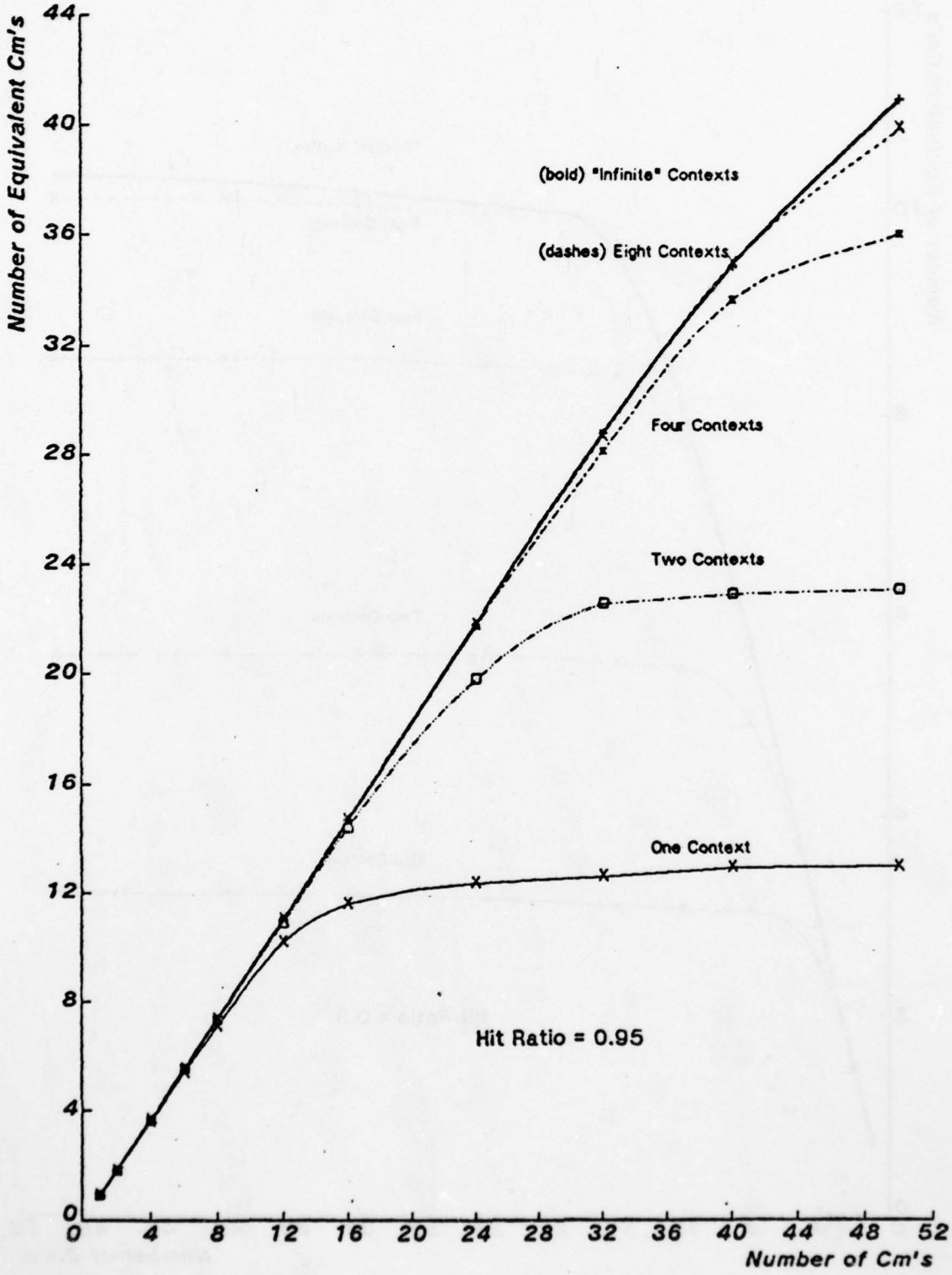


Fig. 3.17: Simulation Model - Effect of # of Contexts on Performance

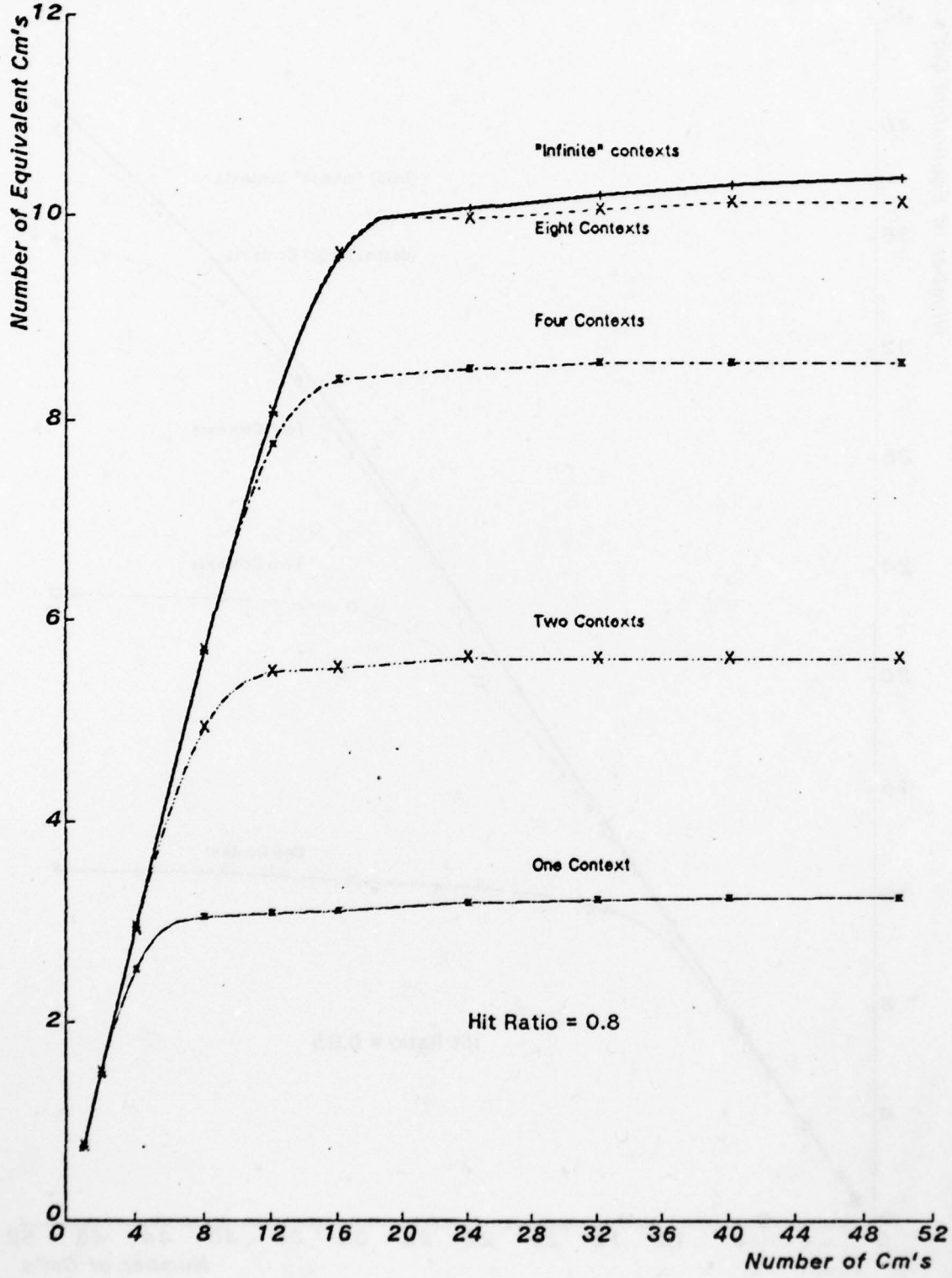


Fig. 3.18: Simulation Model, Effect of # of Contexts on Performance

3.4.2.2 Using faster Processors and memories

As an alternative to the LSI-11 processors used in the current Cm* configuration we wanted to investigate the effect of using faster processors and memories, but retain same Kbus and Pmap speeds. Fig. 3.19 and 3.20 show the results of comparing the current processor (*regular* processor in the figure) with processors three times faster (*X 3 processor* in the figure) - using the same memory and somewhat faster LSI-11 bus (due to faster processor clock), and comparing with a pair of processor and memory which are three time faster (*Pc-Mem X3* curves).

Fig. 3.19 shows the results with *share ratio* = 0 and *hit ratio* of 90% and 95%. As expected the asymptotic throughput saturation of the curves (due to the use of the same Kbus) are identical, but faster processors and memories resulted in a throughput saturation with fewer Cms. E.g. in the 12 Cm's range and a *hit ratio* of 90%, the faster processor gave about 60% increase in throughput while the three times faster Pc-Mem pair gave about a 95% increase in throughput. For a *hit ratio* of 95% the results are an increase of about 80% and 160% in throughput, respectively.

Fig. 3.20 is similar for *share ratio* of 1 (i.e. all external references are to a one memory module). Here we see the effect of using faster memories (Pc-Mem. pair X3 curve), the asymptotic throughput saturation of which is about 60% higher than the regular curve (but still due to the sharing of the memory module). The faster processor asymptotic saturation is slightly lower than that of the regular processor due to the inaccuracy in the computation of the memory interference (the local processor interfering with the external DMA access, as explained before).

From these figures we can see the potential that still exist in the current system for using faster processors and memories in a single cluster. A significant increase in performance would result, even without changing any of the other components of the system.

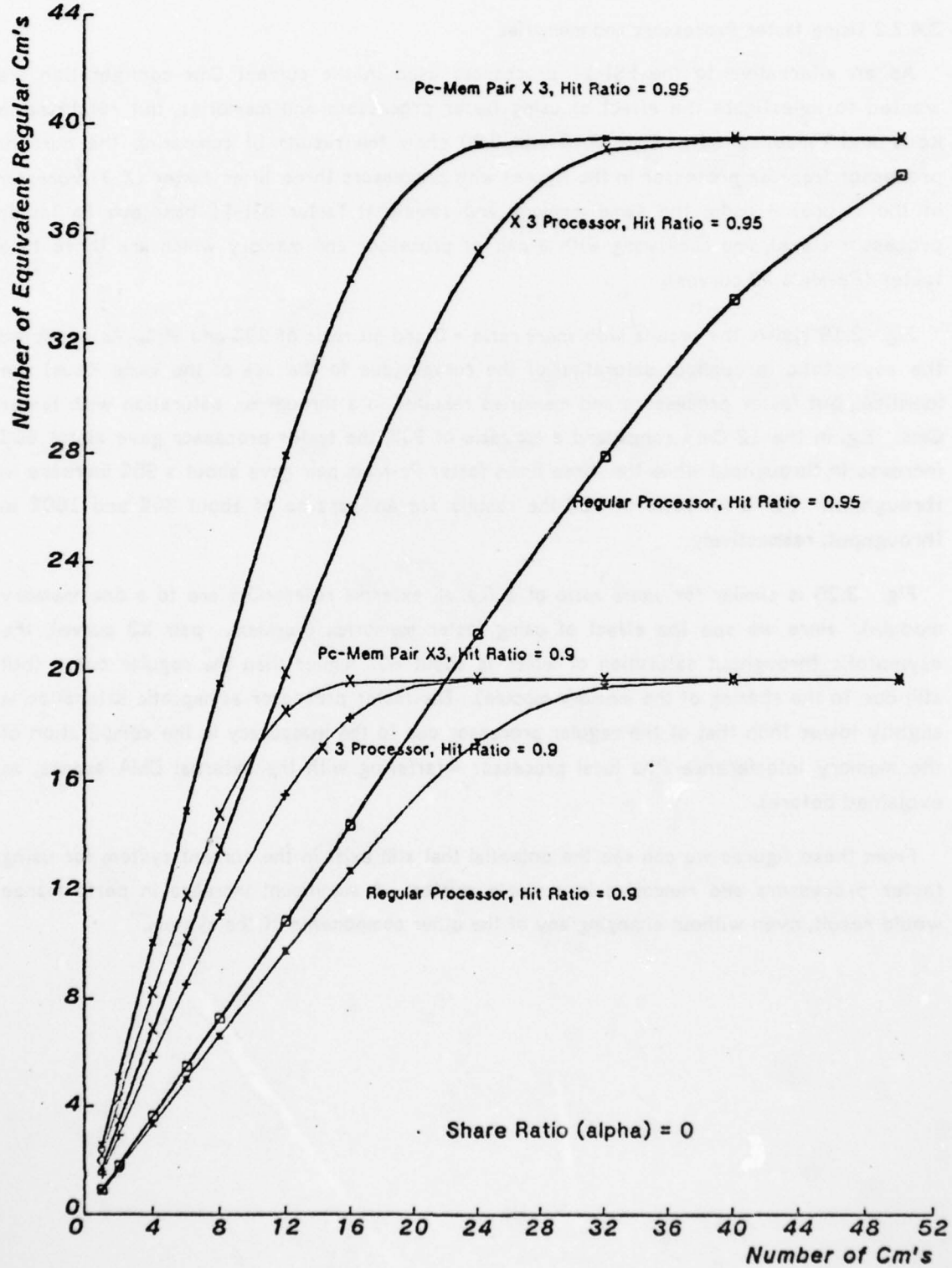


Fig.3.19: Network Model, 1 Cluster, Effect of Processor/Memory Speed

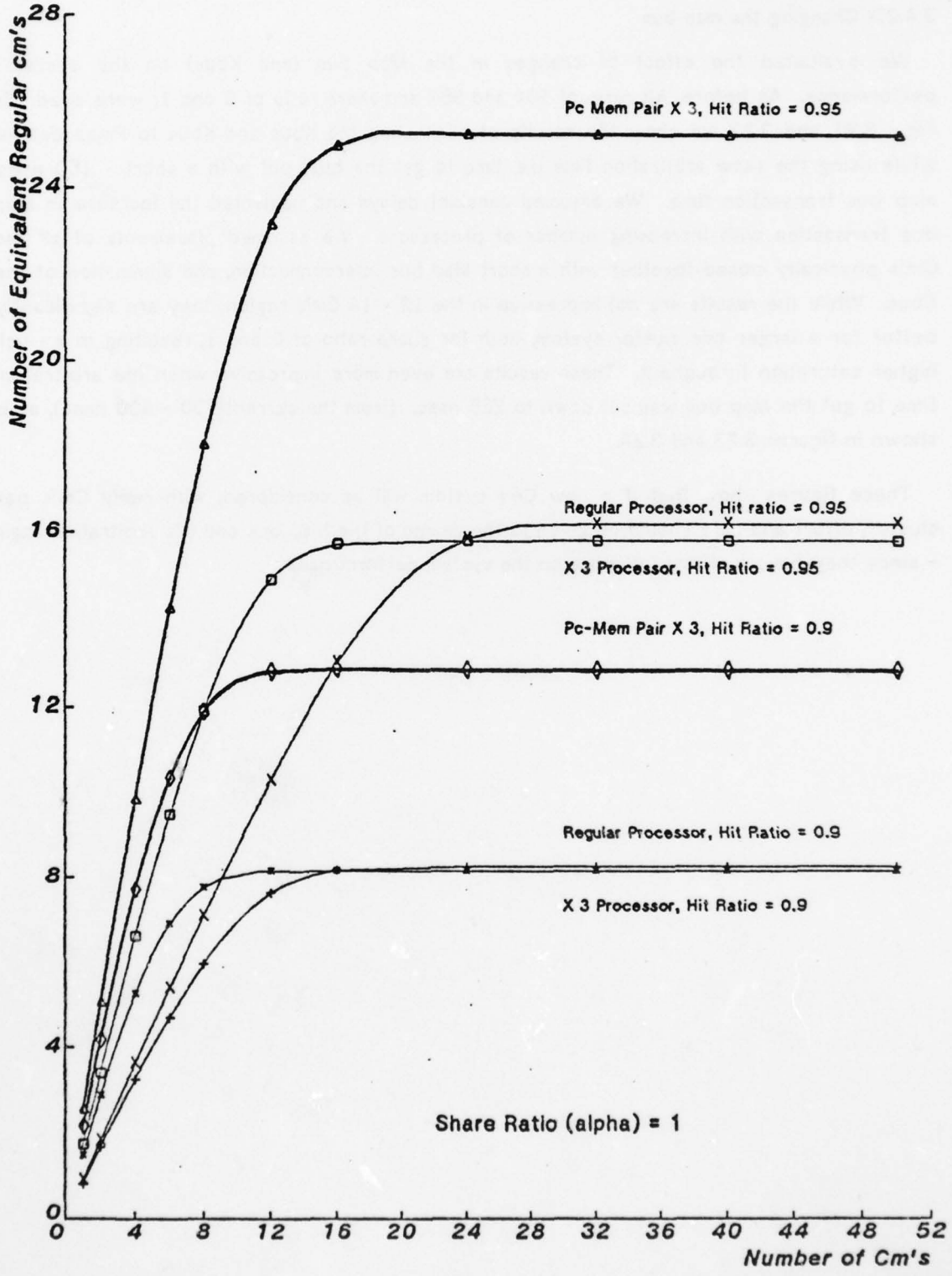
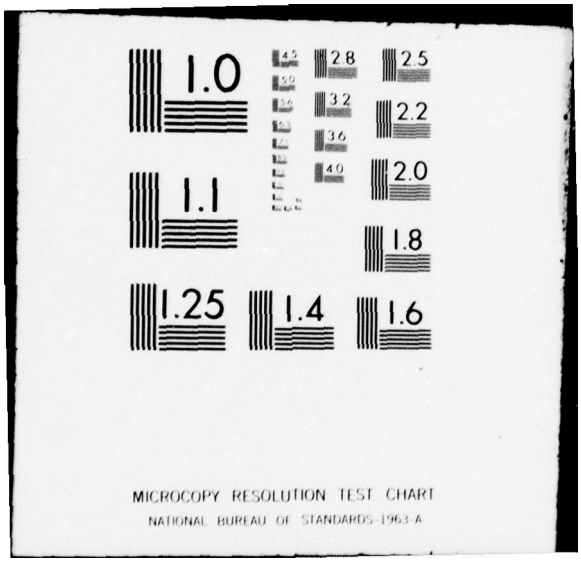


Fig.3.20: Network Model, 1 Cluster, Effect of Processor/Memory Speed

AD-A060 495 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
PERFORMANCE EVALUATION OF MULTIPLE PROCESSOR SYSTEMS.(U)
AUG 78 L RASKIN F44620-73-C-0074
UNCLASSIFIED CMU-CS-78-141 NL

2 OF 3
AD
A060495


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

3.4.2.3 Changing the map bus

We evaluated the effect of changes in the Map bus (and Kbus) on the system's performance. As before, *hit ratio* of 90% and 95% and *share ratio* of 0 and 1, were used. In Fig. 3.21 and 3.22 we show the results of eliminating the Kbus and Kbus to Pmap delays, while using the same arbitration time (i.e. time to get the bus) but with a short - 150 nsec. map bus transaction time. We assumed constant delays and neglected the increase in Map bus transaction with increasing number of processors. We assumed placements of all the Cm's physically closed together with a short Map bus interconnection, and elimination of the Kbus. While the results are not impressive in the 12 - 14 Cm's region, they are significantly better for a larger one cluster system, both for share ratio of 0 and 1, resulting in a much higher saturation throughput. These results are even more impressive when the arbitration time to get the Map bus was cut down to 225 nsec. (from the current 700 - 900 nsec), as is shown in figures 3.23 and 3.24.

These figures show that if a new Cm* system will be considered, with many Cm's per cluster, effort and care should be given to the design of the Map bus and it's arbitration logic - since they have a strong influence on the system performance.

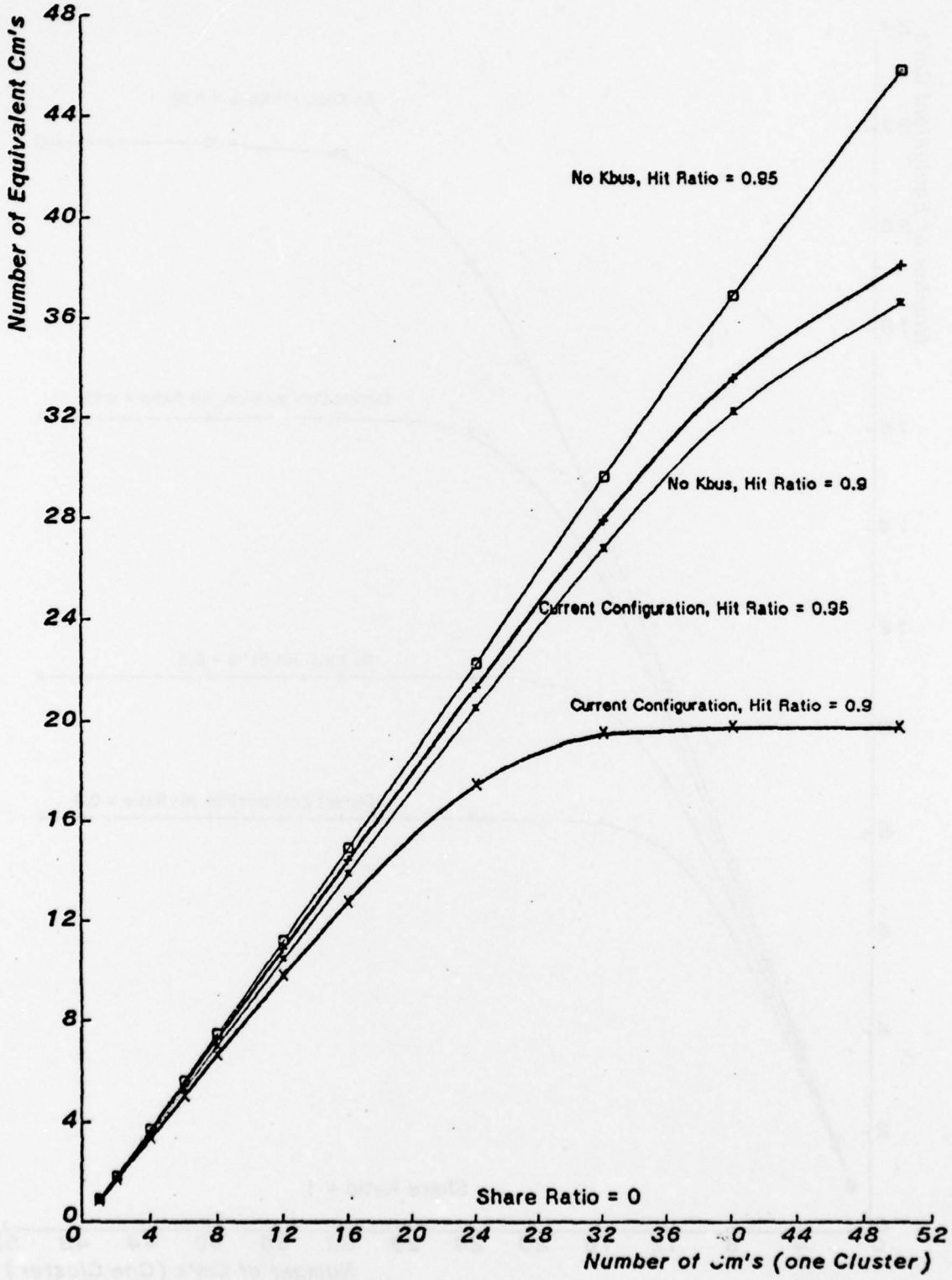


Fig. 3.21: No Kbus, Same Arbitration, 150 nsec Map Transaction

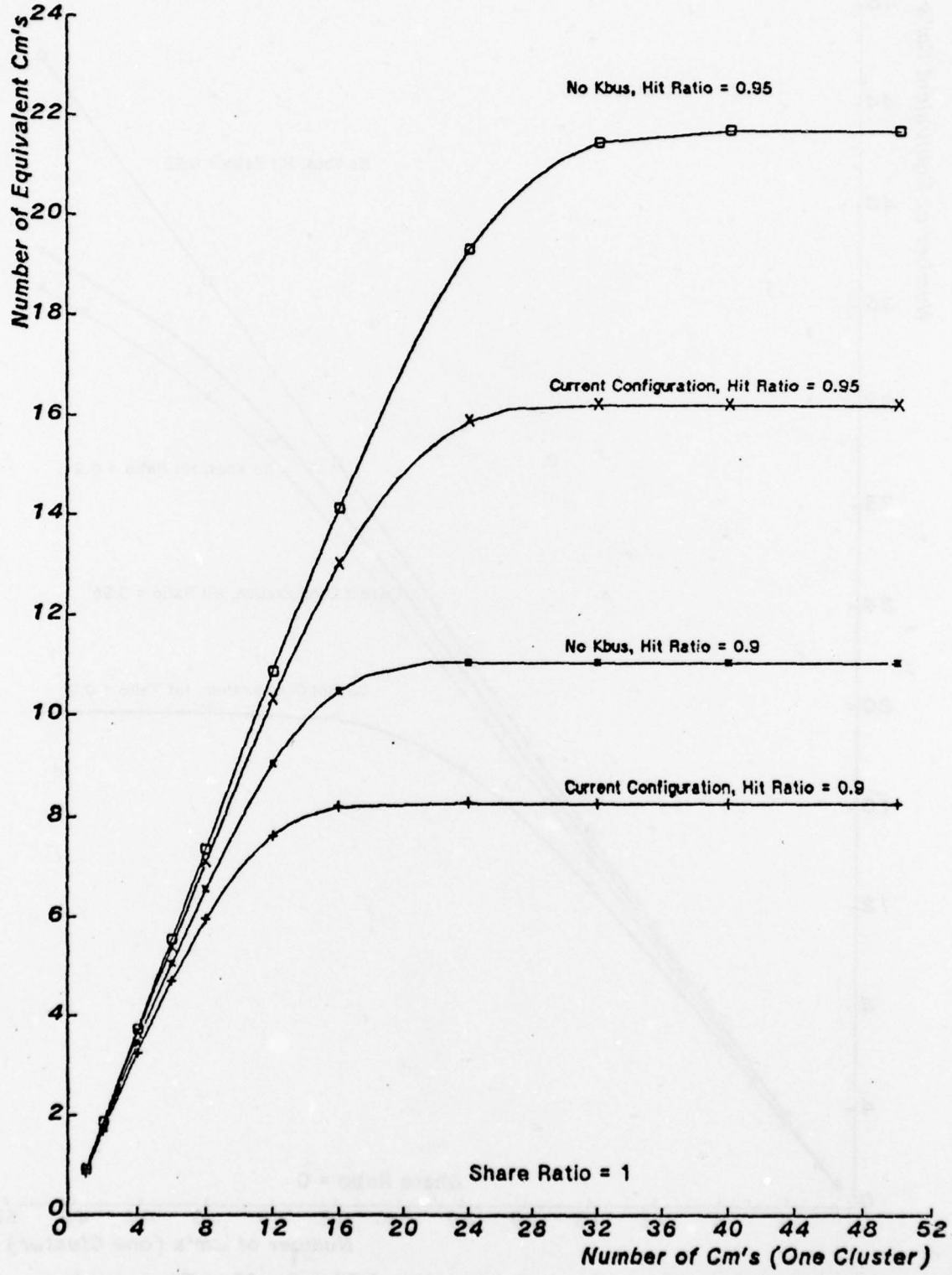


Fig. 3.22: No Kbus, Same Arbitration, 150 nsec Map Transaction

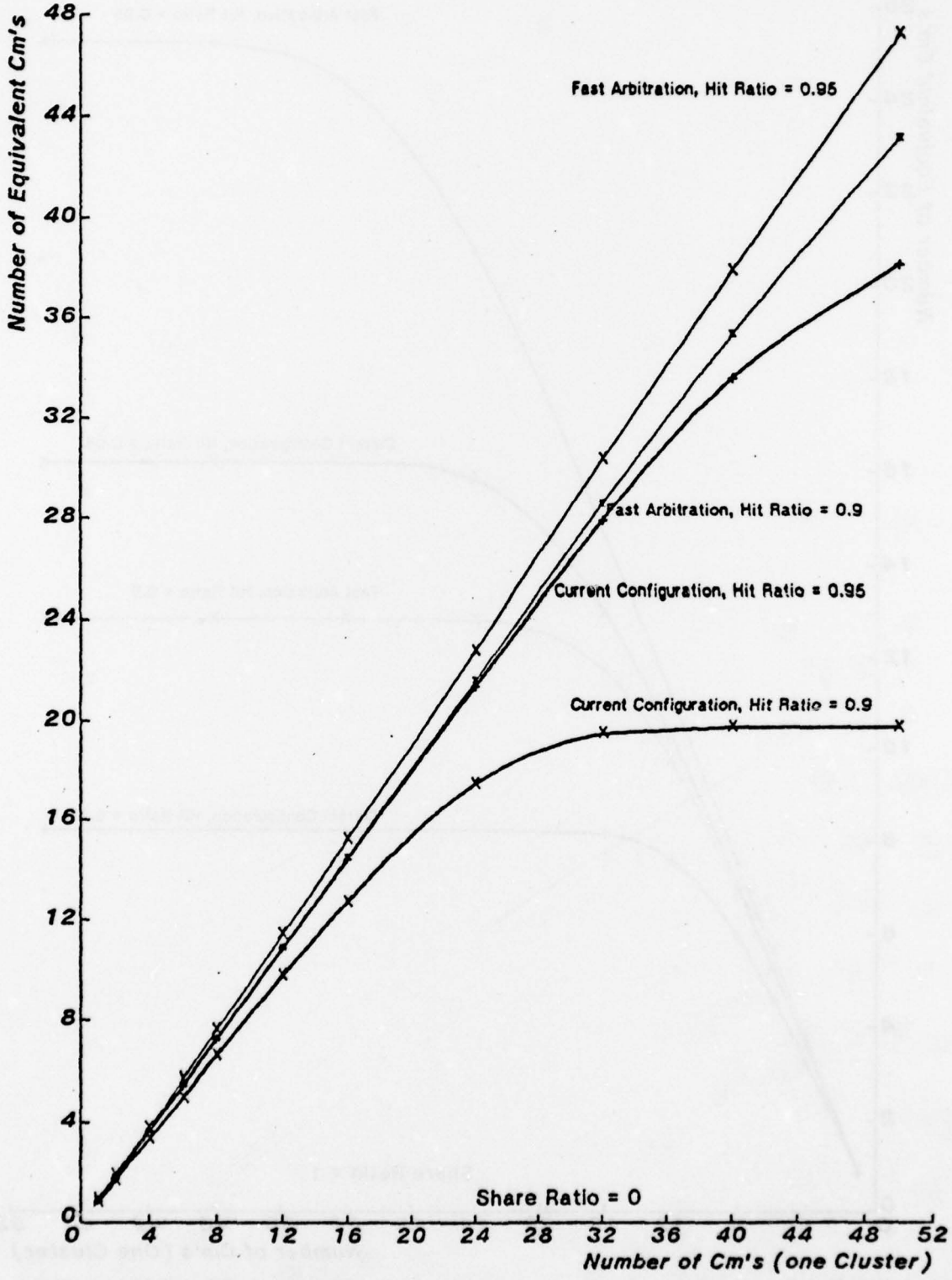


Fig.3.23: No Kbus, Arbitration = 225 nsec, 150 nsec Map Transaction

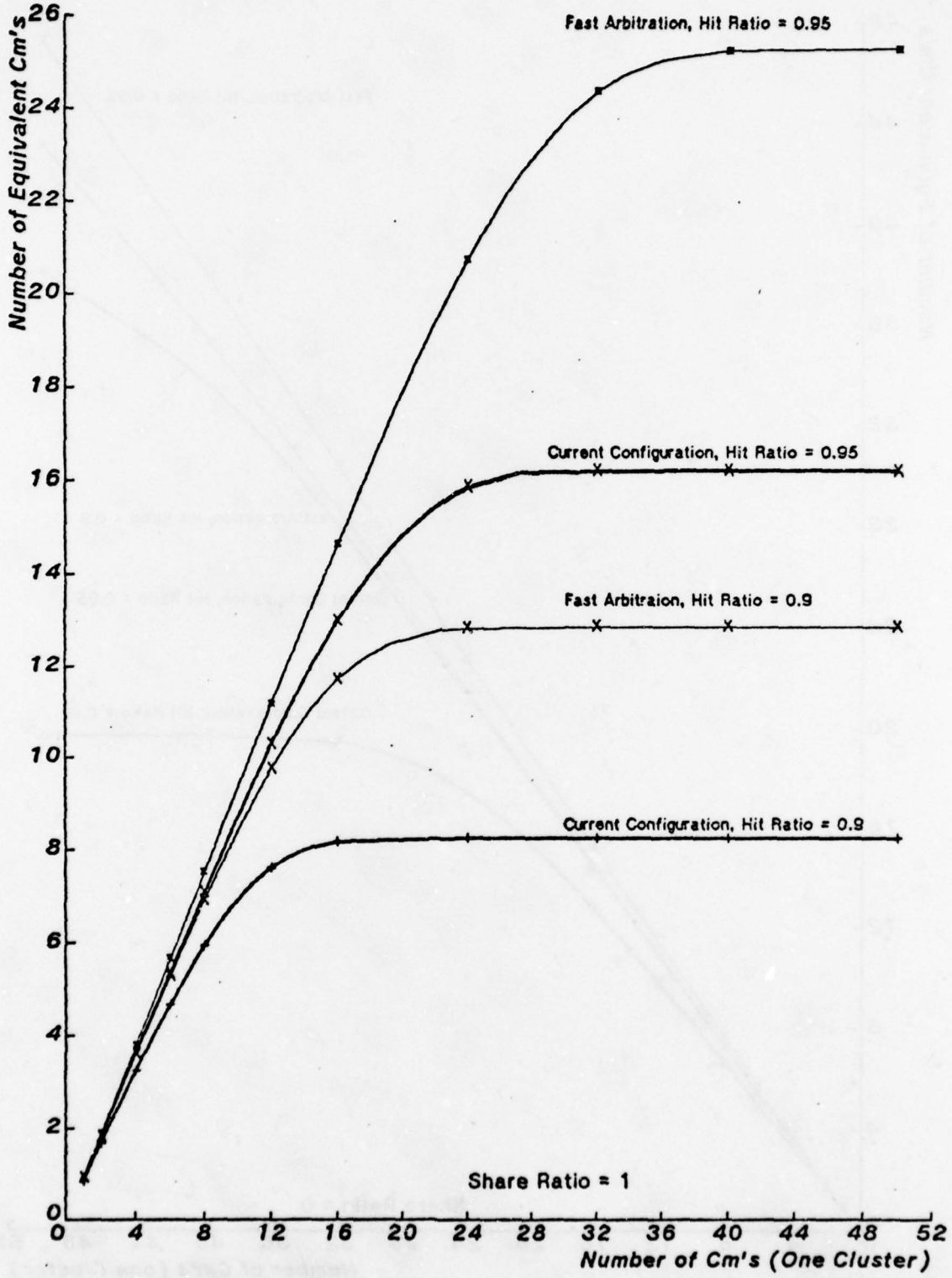


Fig.3.24: No Kbus, Arbitraion = 225 nsec, 150 nsec Map Transaction

3.4.2.4 Changing Kmap clock rates

In this paragraph the effect of changing the main clock that determines the cycle time of the Kbus and the Pmap, is evaluated. This evaluation was motivated by possible need to slow down the Pmap clock to accommodate larger memory I.C's and the possible use of a faster Kbus in the current configuration.

Fig 3.25, 3.26, 3.27 and 3.28 show the throughput results, as before for *hit ratios* of 90% and 95% and *share ratio* of 0 and 1. The five curves show the throughput results of the current system (Kbus cycle time of about 104 nsec. and Pmap cycle time of 157 nsec.) compared to the following clock rates:

(a) Kbus - 100 nsec, Pmap - 200 nsec.

(b) Kbus - 90 nsec., Pmap - 180 nsec.

(c) Kbus - 80 nsec., Pmap - 160 nsec.

(d) Kbus - 80 nsec., Pmap - 80 nsec. (currently not a possible change).

The results show clearly the sensitivity of the throughput performance to changes in the Kbus clock. For *share ratio* of 0 (i.e. no memory interference) the saturation curve is about 10% higher for each 10 nsec (about 10%) decrease in Kbus clock rate (this is expected knowing that the Kbus is the system's bottleneck in this case). The results are insensitive to changes in the Pmap clock - which generally is not a bottleneck in a one cluster system (not considering here special Pmap operations and inter-cluster references). Similar observations were reported in Chapter 2. The throughput is less sensitive to changes in clock rates in the case of *share ratio* of 100% - when the memory module is the system's bottleneck.

These results show again that the Kbus clock should be tuned to be as low as possible if a larger, one Cm* cluster, is considered. Changes in the Pmap clock are insignificant.

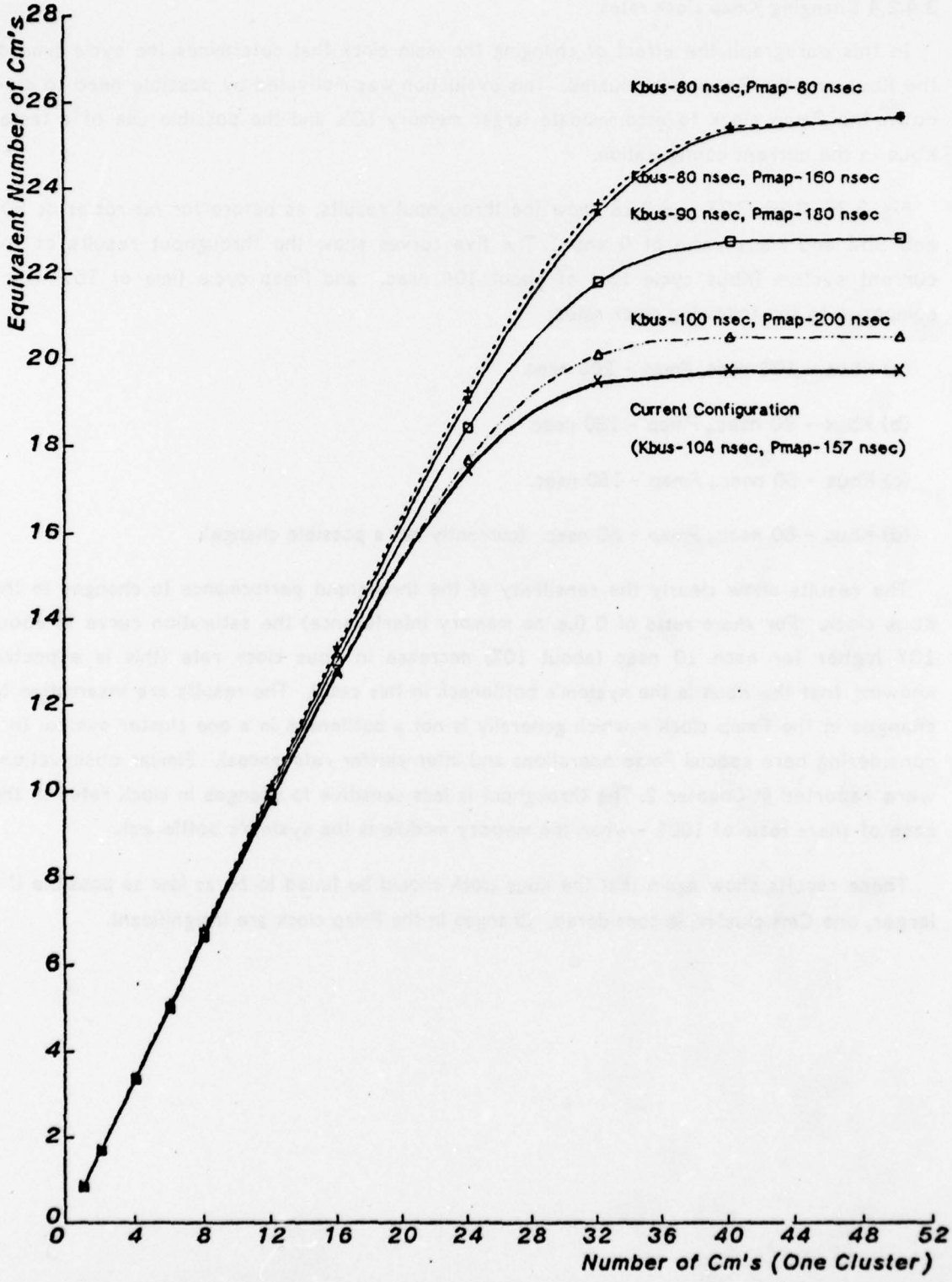


Fig. 3.25: Changing Kmap Clock, Hit Ratio = 0.9, Share Ratio = 0

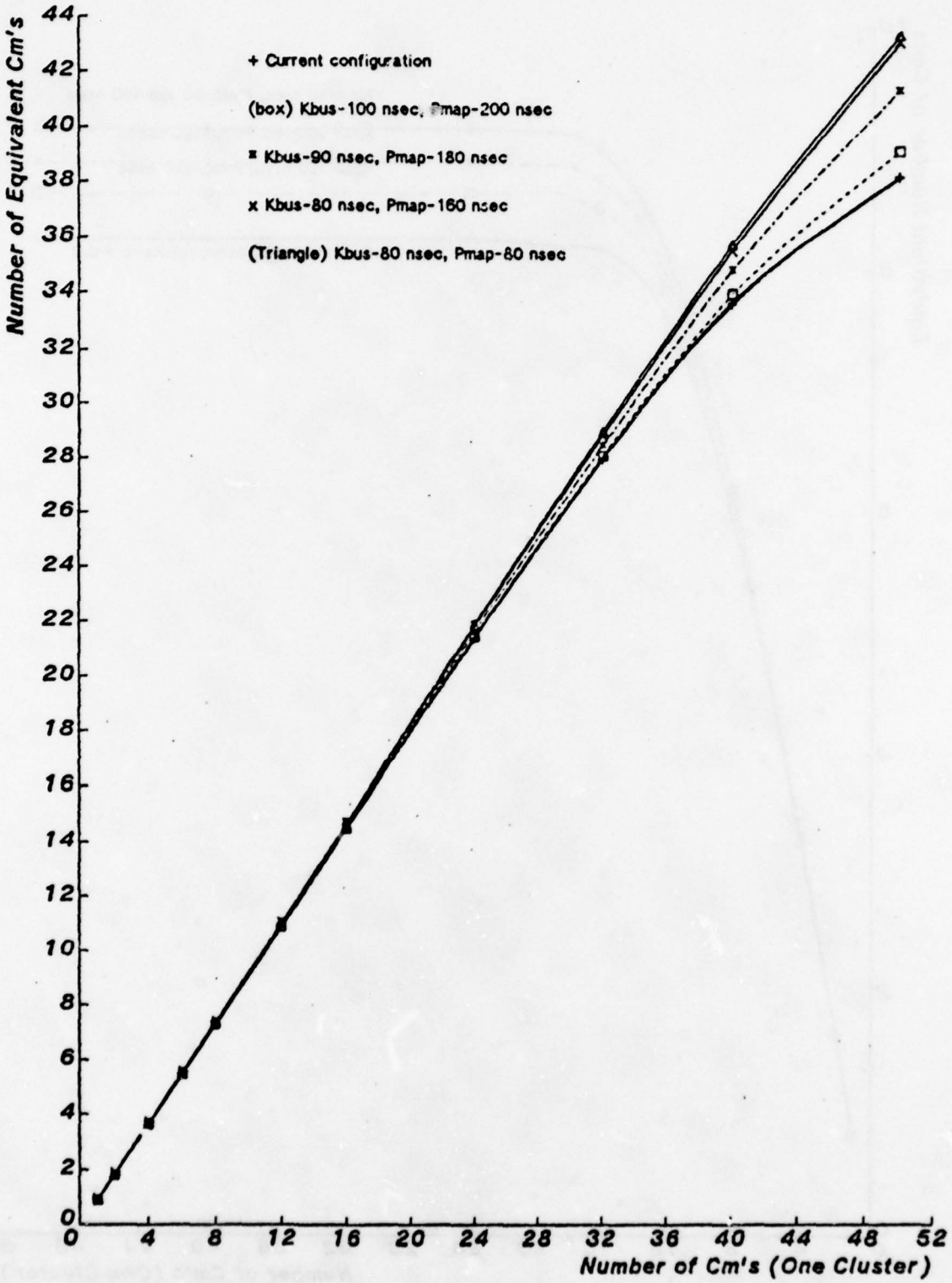


Fig. 3.26: Changing Kmap Clock, Hit Ratio = 0.95, Share Ratio = 0

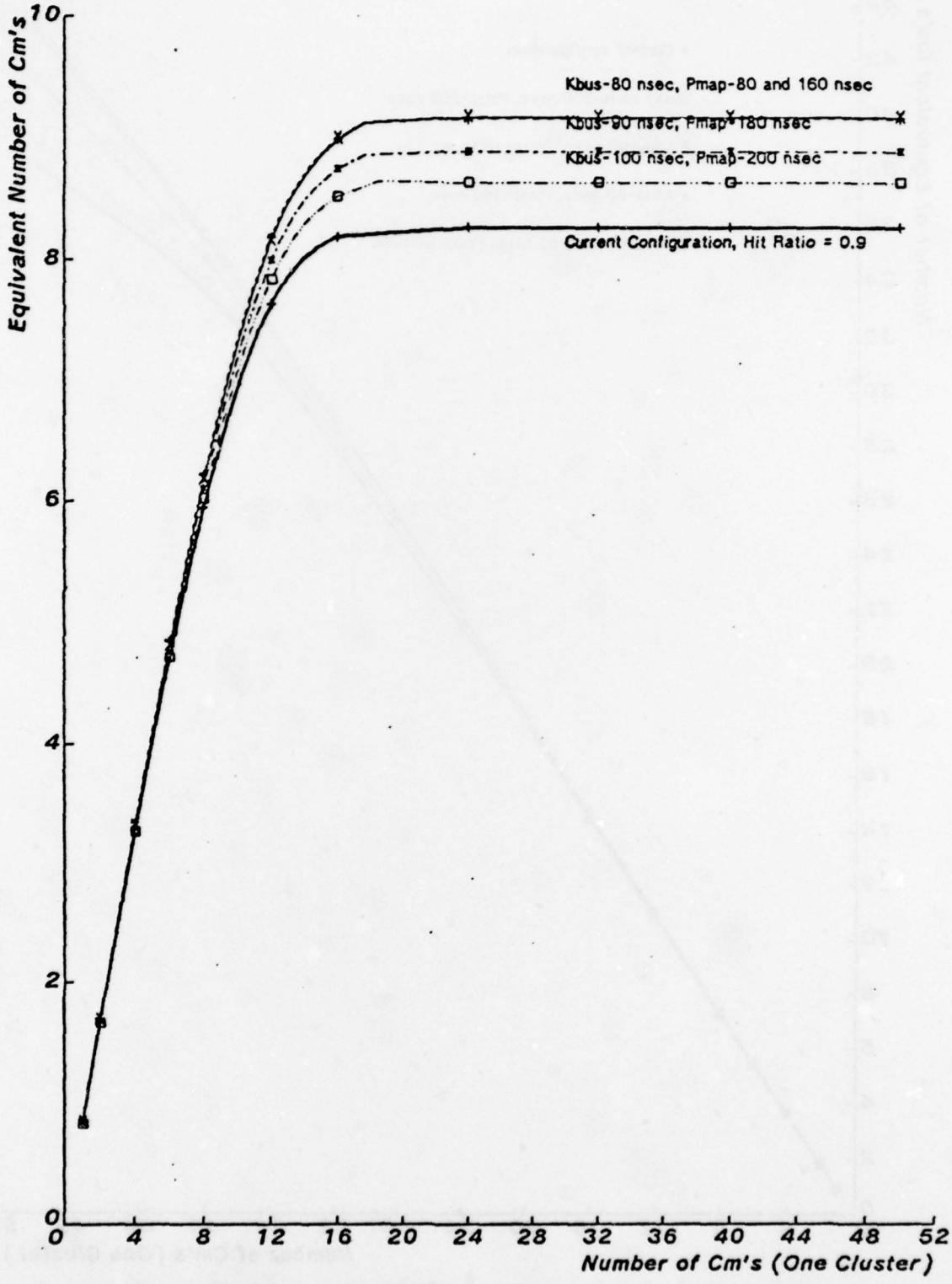


Fig. 3.27: Changing Kmap Clock, Hit Ratio = 0.9, Share Ratio = 1

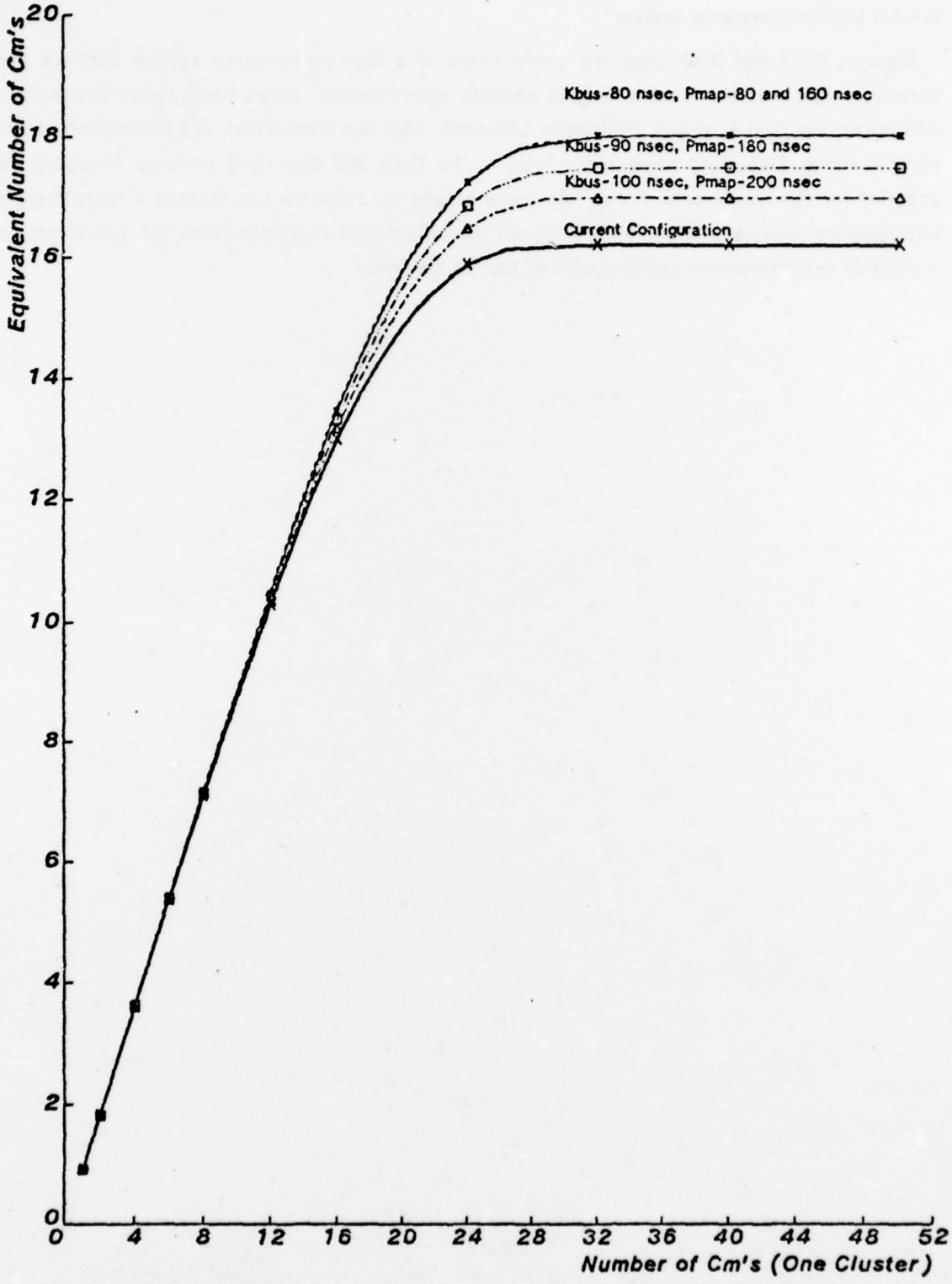


Fig. 3.28: Changing Kmap Clock, Hit Ratio = 0.95, Share Ratio = 1

3.4.2.5 High performance system

Figures 3.29 and 3.30 show the performance of a high performance system that we can foresee - one cluster incorporating all possible improvements: three times faster processors and memories, fast Map bus arbitration, 150 nsec. Map bus transaction, and elimination of the central Pmap (assuming some extra logic in the Cm's and operating systems functions to account for the Pmap operations). As these graphs illustrate we can foresee a large margin between the current design and the (crude) theoretical limit of performance for a new design - roughly a performance improvement of four to six times.

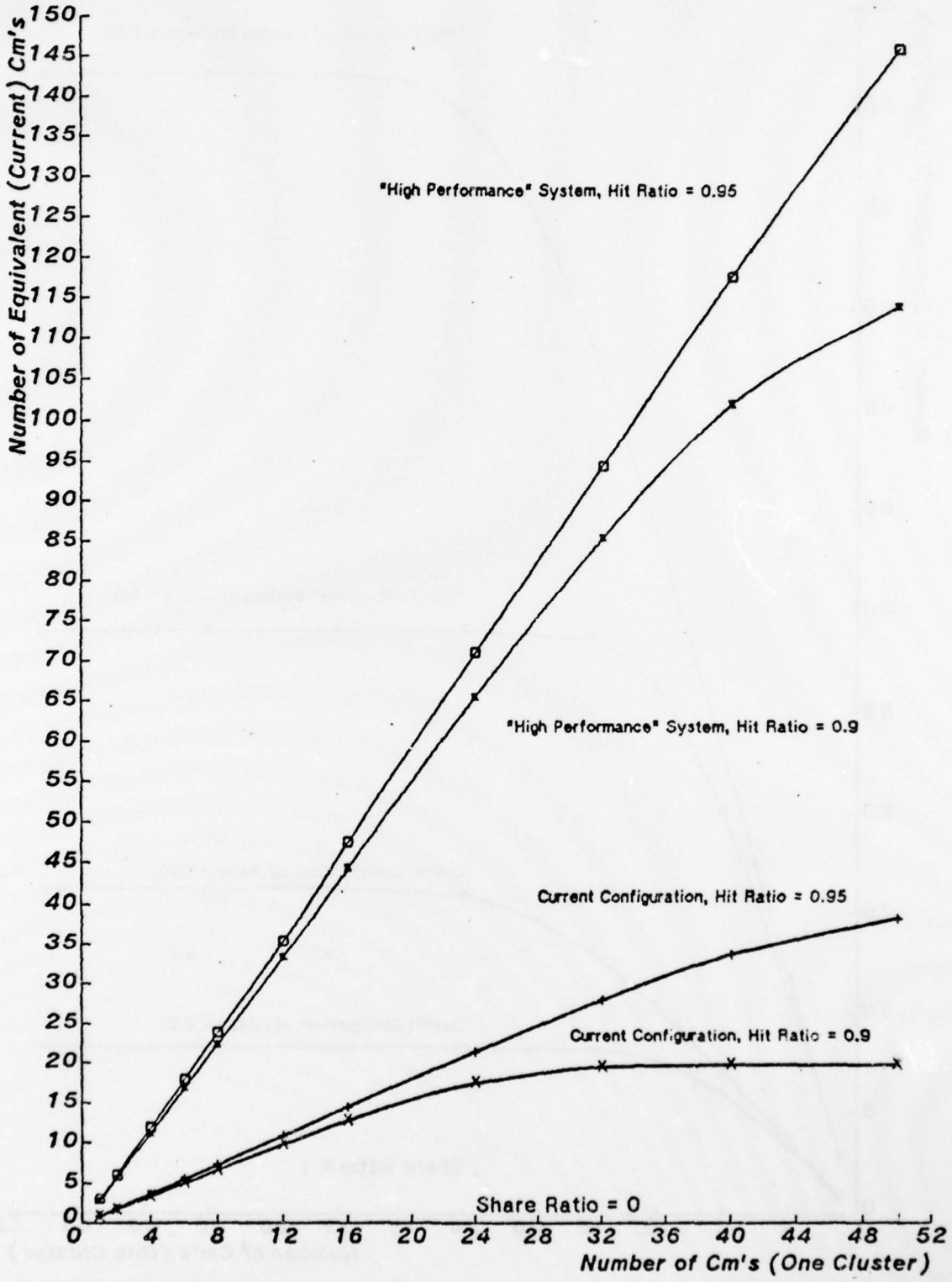


Fig. 3.29: High Performance, No Kbus-Pmap, Fast Pc.-Mem. and Arbit.

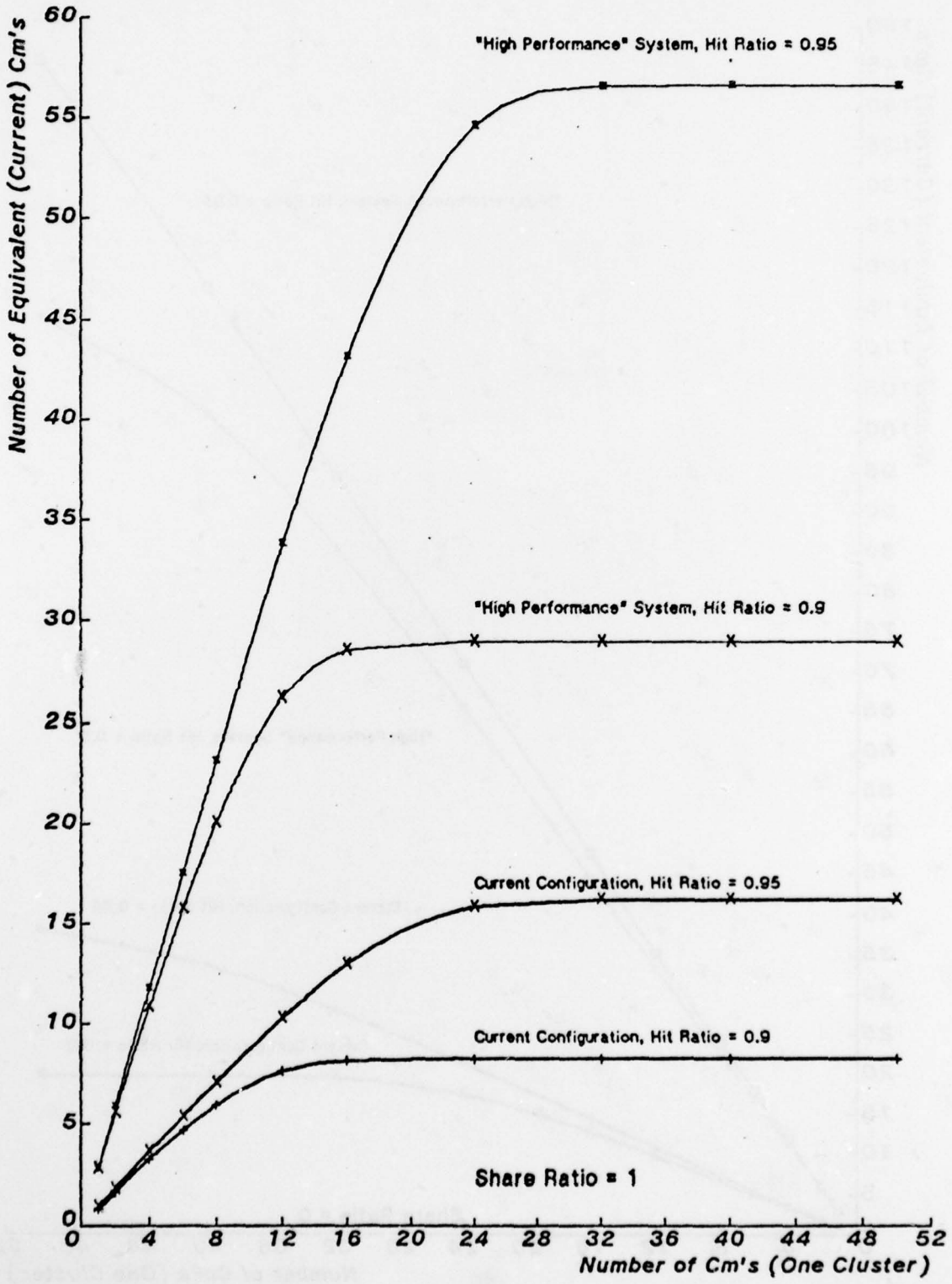


Fig. 3.30: High Performance, No Kbus-Pmap, Fast Pc.-Mem. and Arbit.

3.4.3 Performance of a Multi-Cluster System

In this sub-section we present the results of evaluating a multi-cluster Cm* system, using the model given in Chapter 3. Here the third layer in the Cm*-like memory hierarchy structure - the inter-cluster communication is considered for the first time.

The main performance issues considered were: optimal distribution of Cm's between clusters to achieve maximum performance, effect of using faster processors on the multi-cluster system performance, and the effect on performance of replacing the parallel inter-cluster bus by a serial line.

3.4.3.1 Comparing one cluster to multi-cluster system

The question considered here is the issue of optimal distribution of an application program in the system. Given a parallel application program, the parameters of which are known, we want to find an optimal way to assign Cm's in a multi-cluster configuration to get the maximum performance out of the system.

The answer to this question is not trivial - as a one cluster system with many Cm's might suffer contention of the central mapping resource, while dividing the application program across many clusters (with less Cm's in each) will cause increased inter-cluster communication rate demands for the relatively slow inter-cluster resources, and thus lower performance.

As explained in Section 3.3 the parameters of the model were: The *hit ratio*, the *share ratio* (Alpha) - to a single memory module in one cluster in the system, and *inter-cluster* ratio (Beta) - i.e. ratio of the remaining external memory references that are referencing a memory in another cluster. In all these experiments we used, again, *hit ratios* of 90% and 95%, and *share ratio* of 0 and 1. We assumed homogeneous distribution of Cm's in the clusters - e.g. for a 48 Cm's system we have compared a one cluster - 48 Cm's system to 2 clusters - 24 Cm's and 4 clusters - 12 Cm's in each systems. Here, again, we didn't consider the increase in communication time with increased number c' Cm's - and assumed constant bus times.

Figures 3.31 and 3.32 show the potential to increase the performance by using several clusters. This shows the limit of performance gain in the multi-cluster case when we assumed a negligible inter-cluster communication (*multi cluster* ratio and *share* ratio of 0). The figures show the higher performance of a of multi-cluster over a single cluster - simply a result of the interference in the central Map bus and Kbus in the one cluster, many Cm's case.

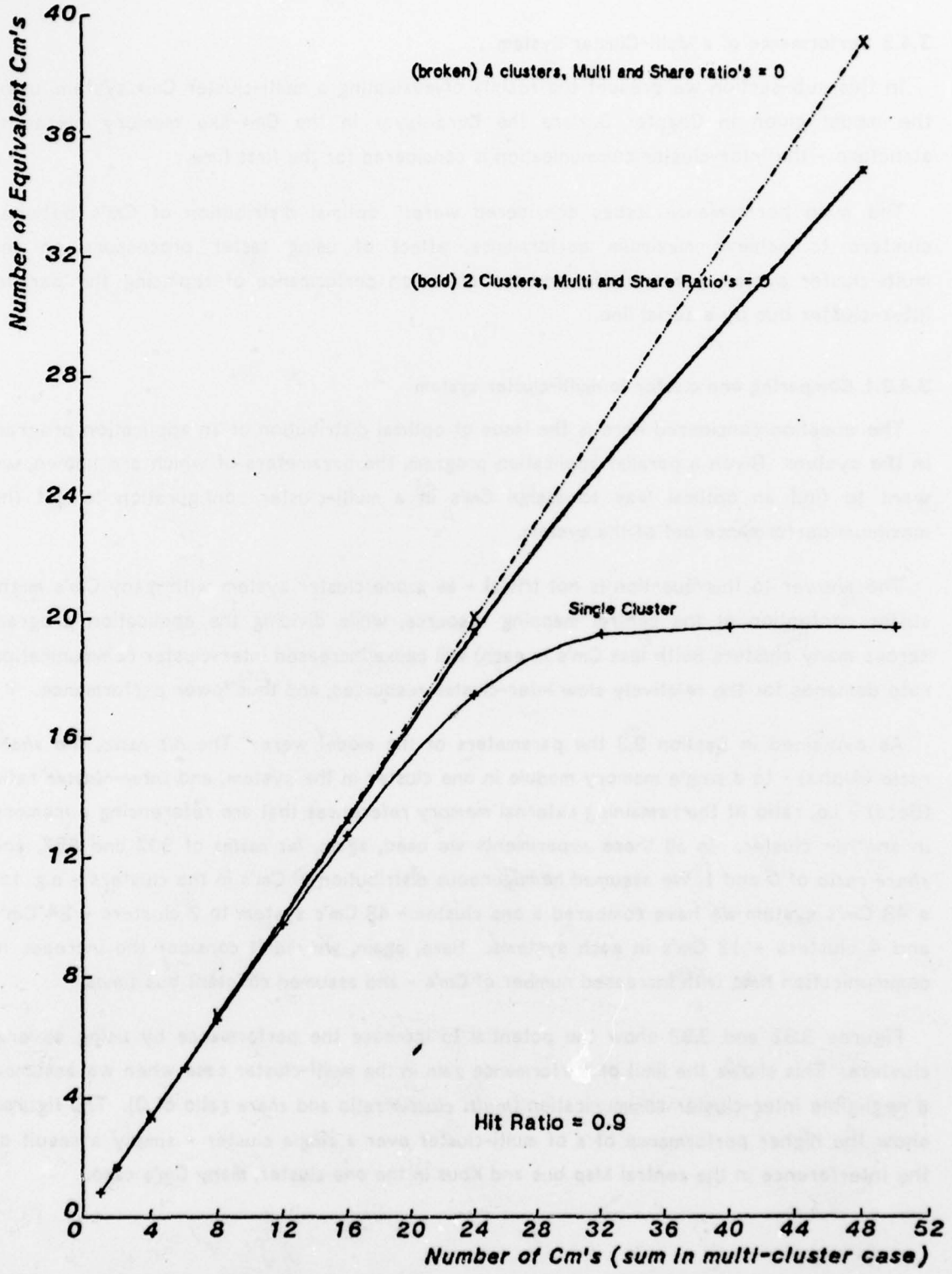


Fig. 3.31: Network Model, Comparing Single and Multi Clusters

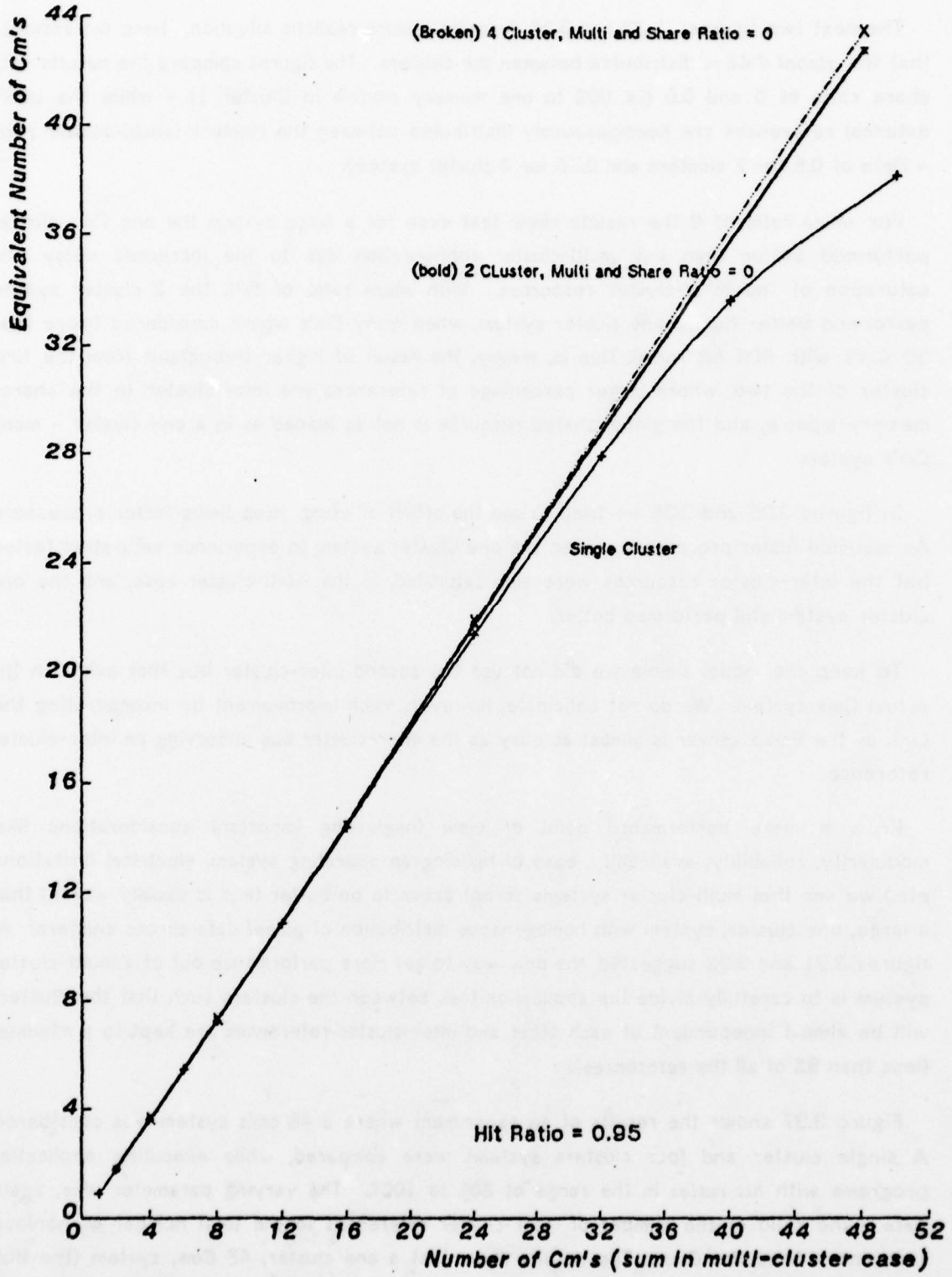


Fig. 3.32: Network Model, Comparing Single and Multi-Clusters

The next two figures: 3.33 and 3.34 describe a more realistic situation. Here we assumed that the *global* data is distributed between the clusters. The figures compare the results with share ratio of 0 and 0.5 (i.e 50% to one memory module in Cluster 1) - while the other external references are homogeneously distributed between the clusters (*multi-cluster* ratio = Beta of 0.5 for 2 clusters and 0.75 for 4 cluster system).

For *share* ratio of 0 the results show that even for a large system the one Cm* cluster performed better than any multi-cluster configuration due to the increased delay and saturation of the inter-cluster resources. With *share* ratio of 50% the 2 cluster system performed better than a one cluster system, when many Cm's where considered (more than 30 Cm's with 90% *hit ratio*). This is, mainly, the result of higher throughput from the first cluster of the two, where larger percentage of references are intra-cluster to the shared memory module, and the global shared resource is not as loaded as in a one cluster - many Cm's system.

In figures 3.35 and 3.36 we tried to see the effect of using three times faster processors. As assumed faster processors caused the one Cluster system to experience saturation faster, but the inter-cluster resources were also saturated, in the multi-cluster case, and the one cluster system still performed better.

To keep the model simple we did not use the second inter-cluster bus that exists in the actual Cm* system. We do not anticipate, however, much improvement by incorporating this Link as the Pmap server is almost as busy as the inter-cluster bus in serving an inter-cluster reference.

From a mere performance point of view (neglecting important considerations like: modularity, reliability, availability, ease of building an operating system, electrical limitations, etc.) we see that multi-cluster systems do not prove to be better (and is usually worse) than a large, one cluster, system with homogeneous distribution of *global* data across clusters. As figures 3.31 and 3.32 suggested, the only way to get more performance out of a multi-cluster system is to carefully divide the application task between the clusters such that the clusters will be almost independent of each other and inter-cluster references are kept to a minimum (less than 5% of all the references).

Figure 3.37 shows the results of an experiment where a 48 cm's system was considered. A single cluster and four clusters systems were compared, while executing application programs with *hit ratios* in the range of 80% to 100%. The varying parameter was, again, *Beta* - the ratio of the number of inter-cluster references to the total number of nonlocal references from the Cms. The results show that a one cluster, 48 Cms, system (the bold

curve) has better performance than a four cluster system with inter-cluster ratios of 50% or higher. This means that only with a good partition of the application program in the four cluster case - such that more than 50% of all the external references are internal within the cluster - can a four cluster system be superior to a single, 48 Cms, cluster. Similar experiments can be made for other configurations.

Given the electrical limitations of the current system it is, nevertheless, encouraging to verify that in many applications, with relatively low *hit ratio*, the multi-cluster system gives reasonably good performance.

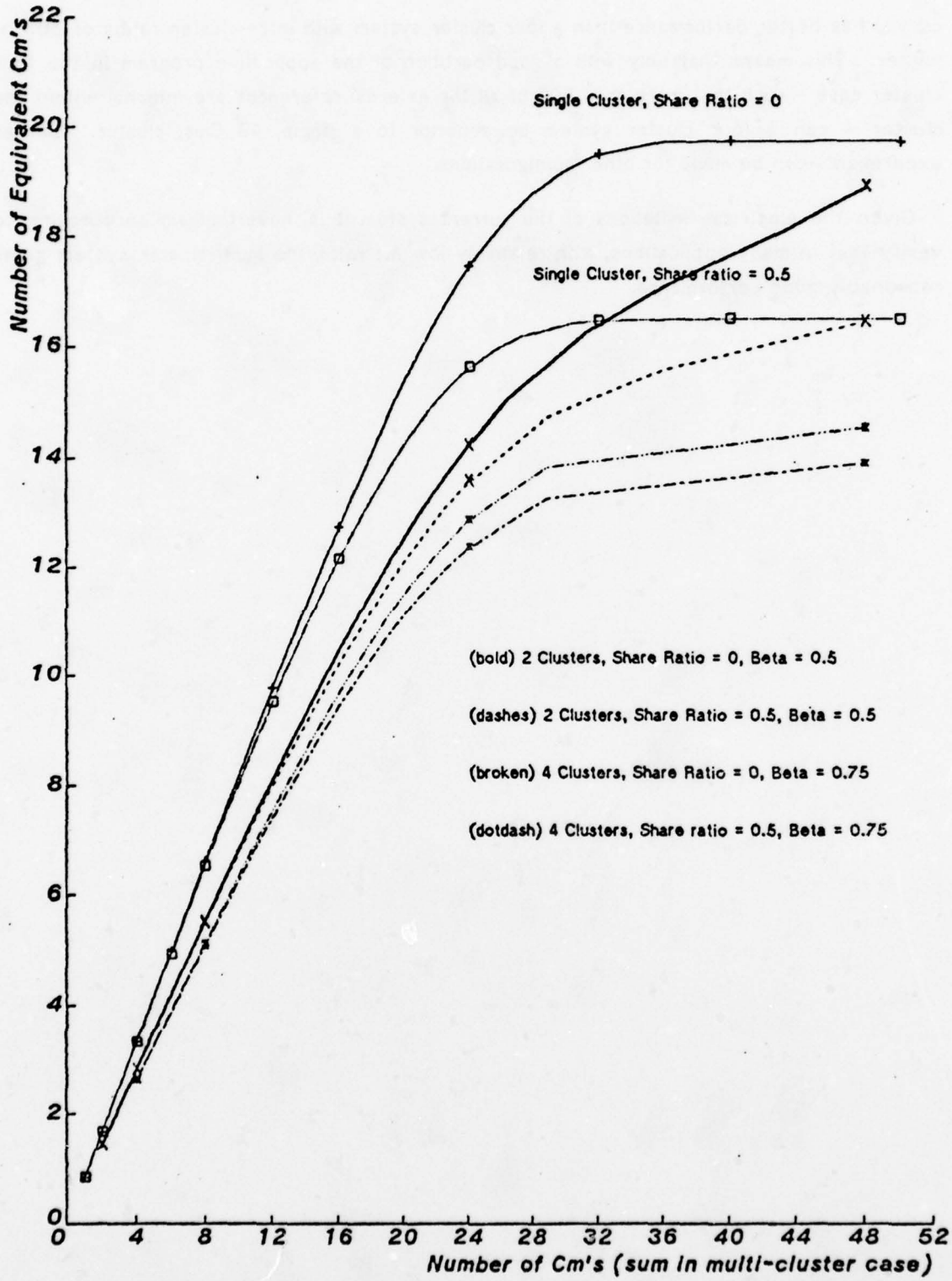


Fig. 3.33: Comparing Single and Multi Clusters, Hit Ratio = 0.9

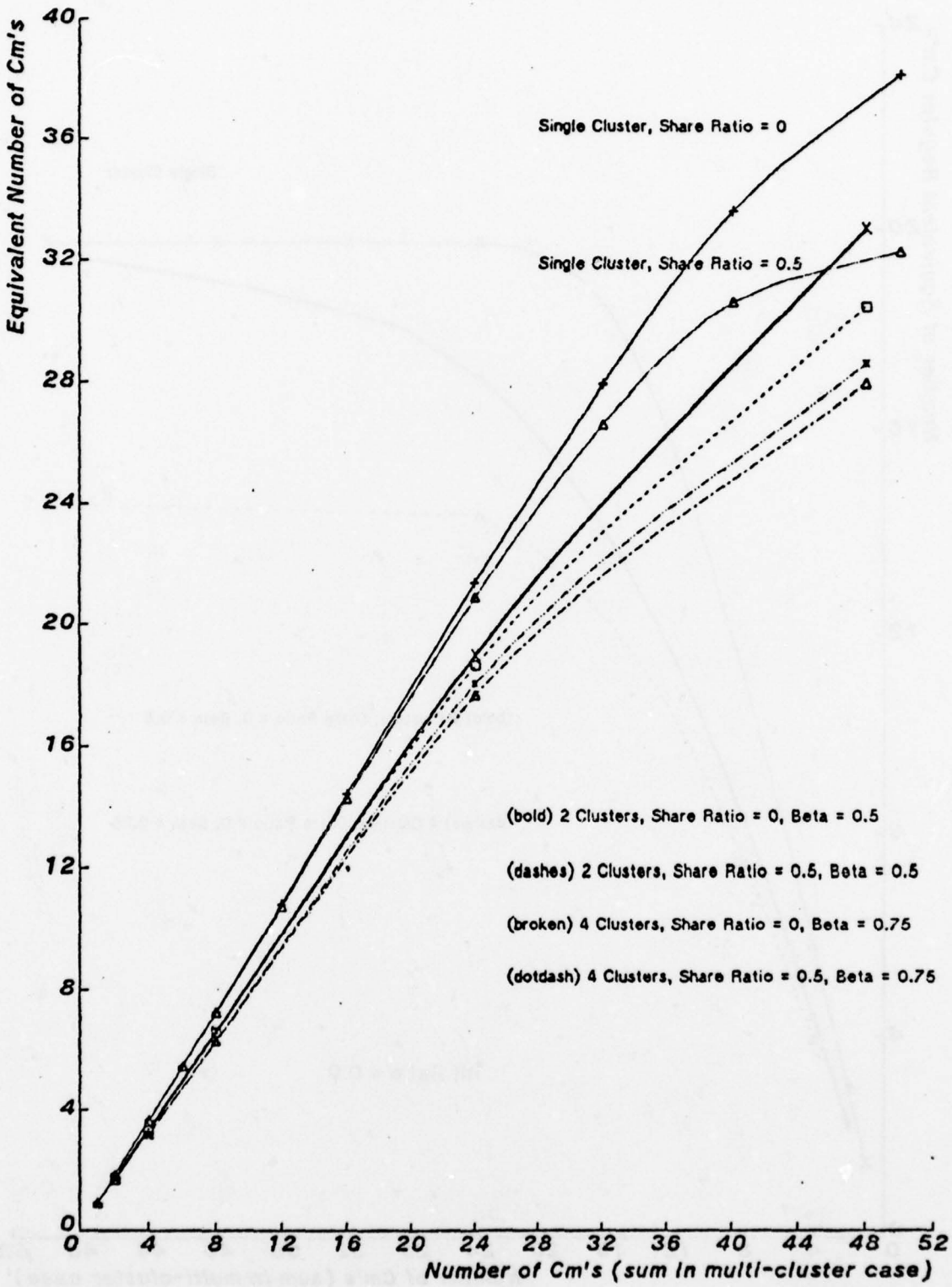


Fig. 3.34: Comparing Single and Multi-Clusters, Hit Ratio = 0.95

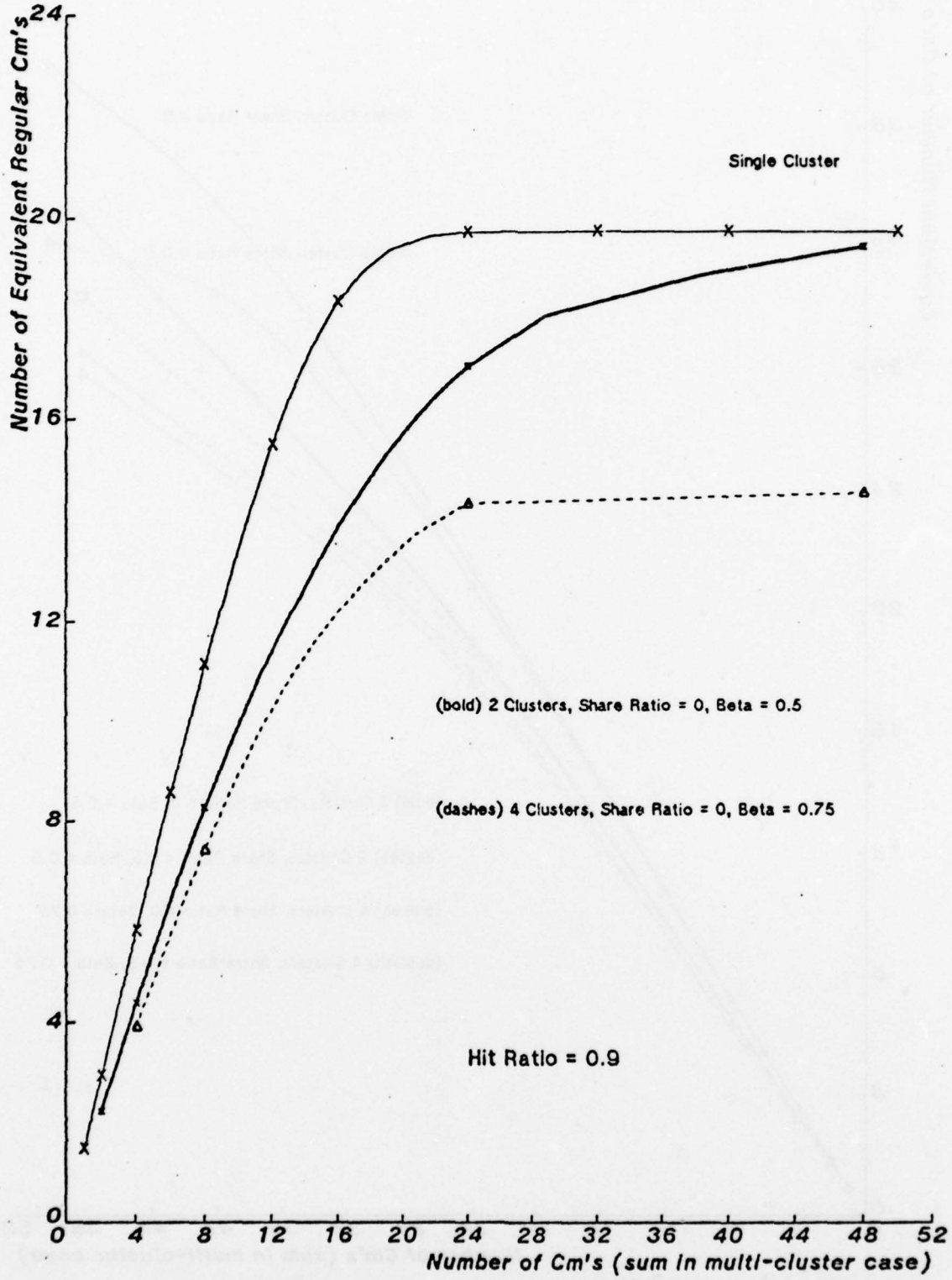


Fig. 3.35: Comparing Single and Multi Clusters, Processor X3 Faster

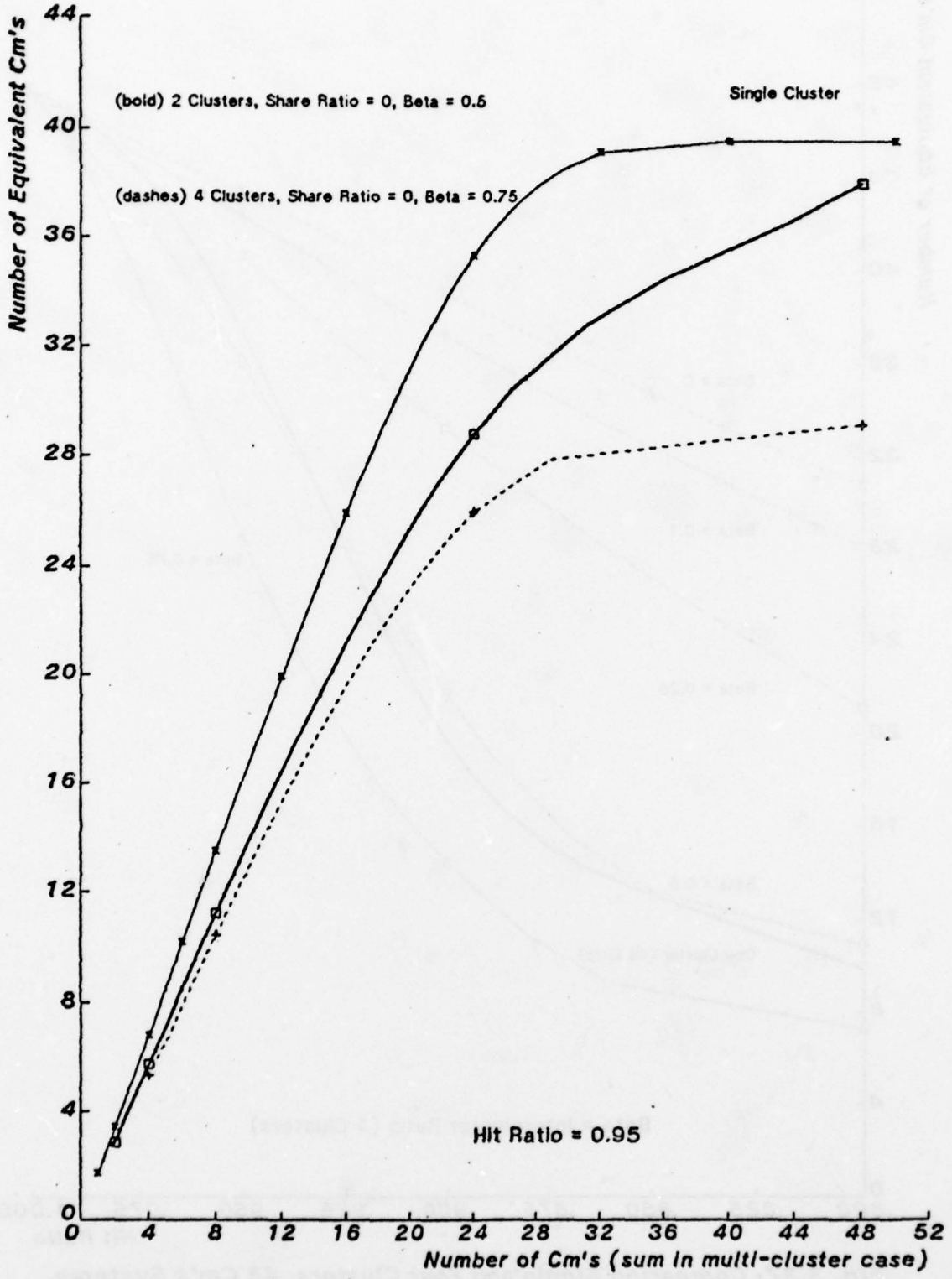


Fig. 3.36: Comparing Single & Multi Clusters, Processor X3 Faster

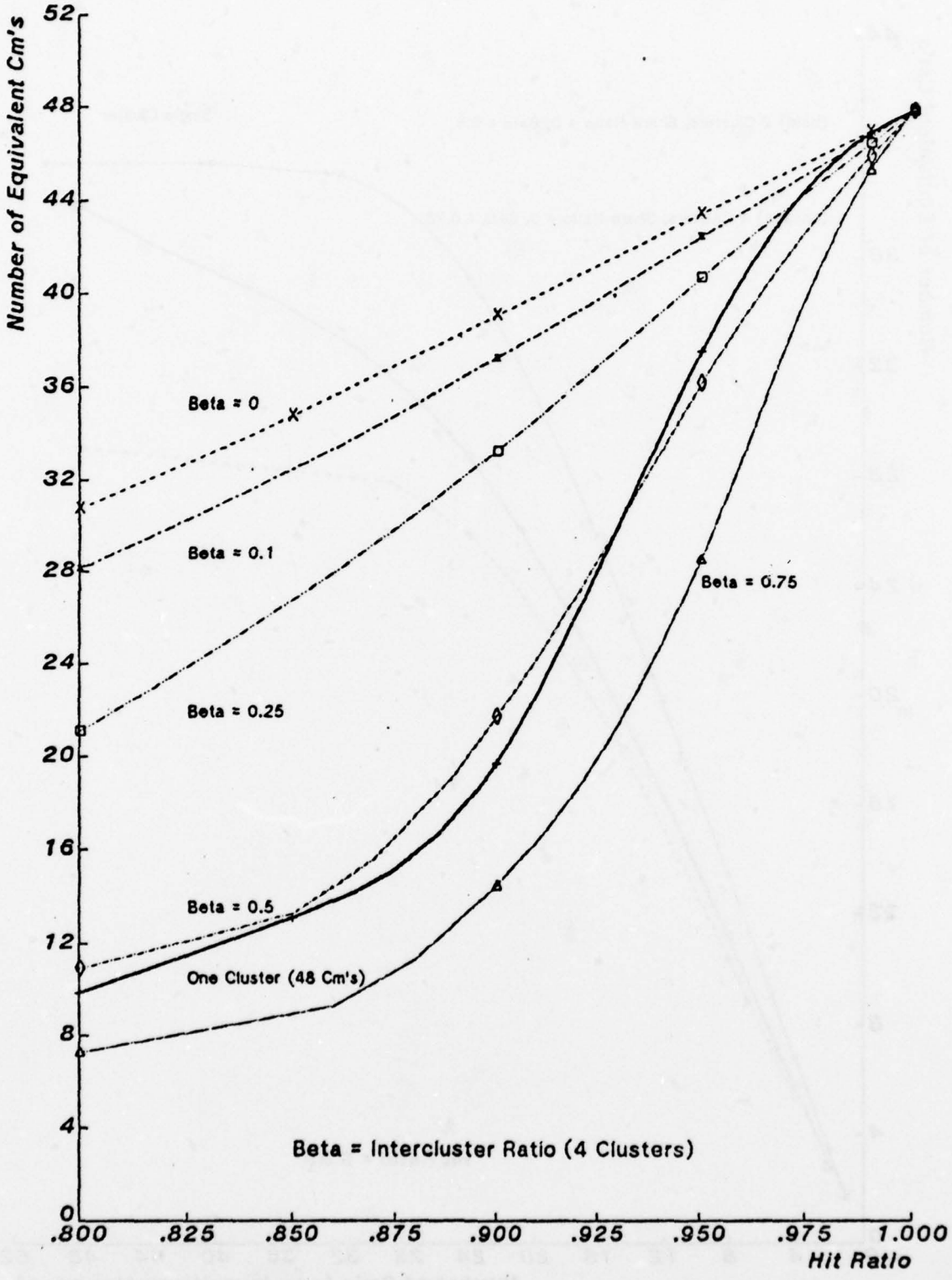


Fig. 3.37: Comparing Single and Four Clusters, 48 Cm's Systems.

3.4.3.2 Changing inter-cluster bus rates

Figures 3.38 and 3.39 show the result of replacing the parallel inter-cluster bus between two cluster by a single serial line. We assumed the same bus arbitration overhead as in the current system. For a word transfer we assumed a protocol similar to the one used in Mil-Std-1553A (multiplex Data bus) - which use a 20 bit, synchronous, Manchester Bi-Phase encoded packet to send a 16 bit word (overhead of 3 bits Sync and 1 bit parity).

As seen in the figures slower rates, with up to 10 Mbits/second cannot compete with the parallel inter-cluster bus. Even for 10 Mbits/second bus rates the inter-cluster bus saturates with about 7 and 11 Cm's in each cluster - for *hit ratio* of 90% and 95%, respectively.

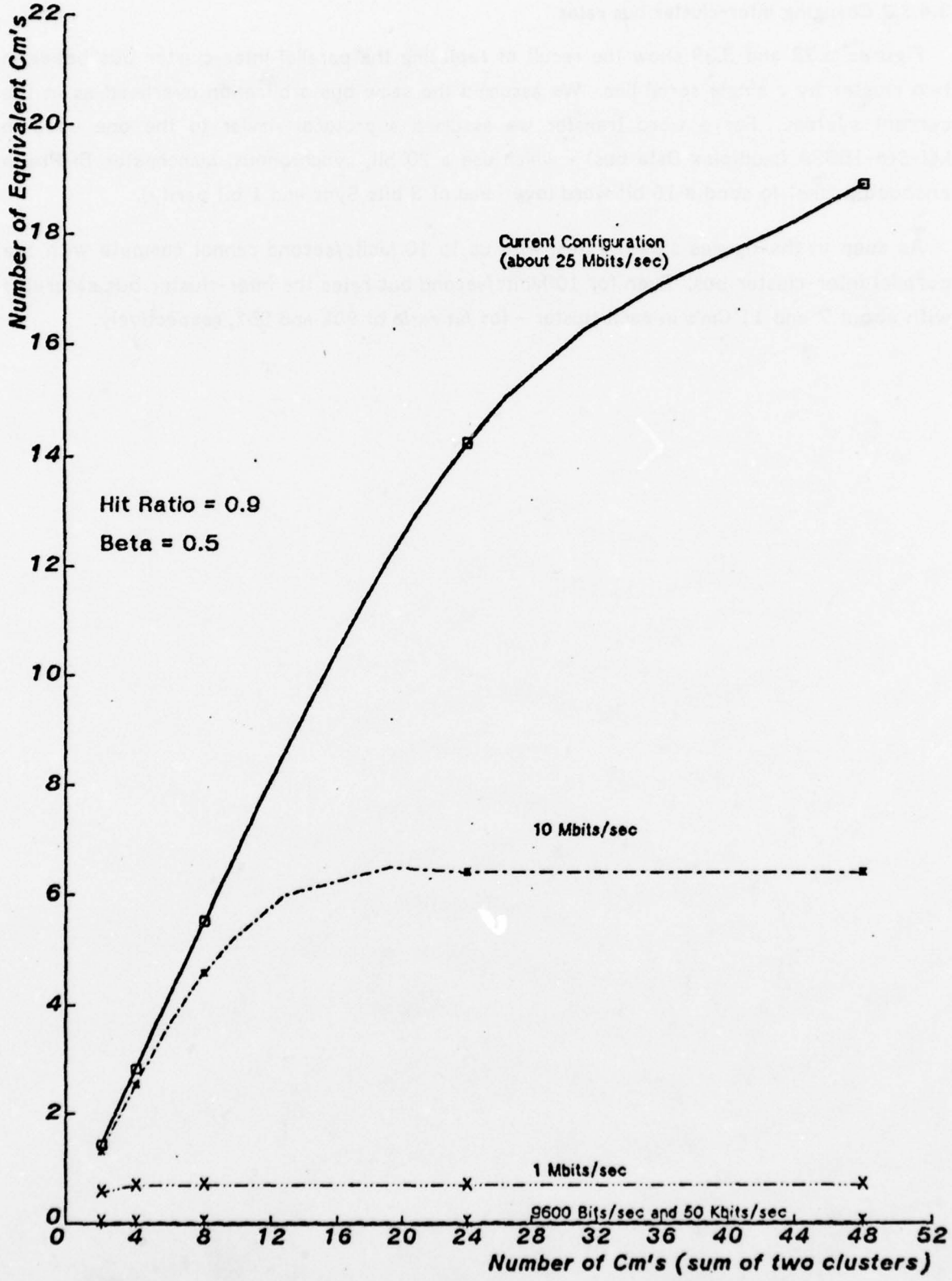


Fig.3.38: Simulating Various Inter-Cluster Rates between 2 Clusters

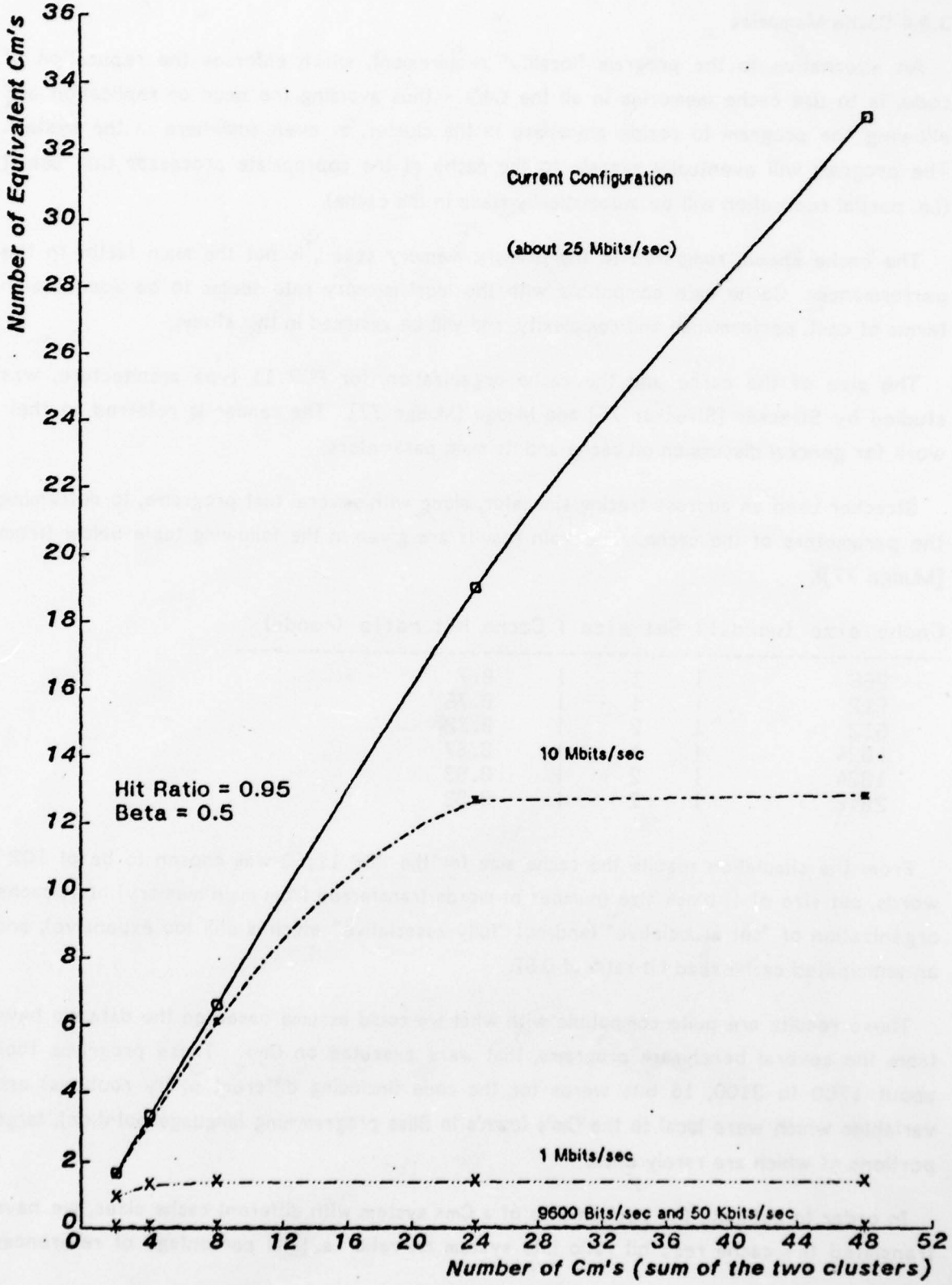


Fig.3.39: Simulating Various Inter-Cluster Rates between 2 Clusters

3.4.4 Cache Memories

An alternative to the program "locality" requirement, which enforces the replication of code, is to use cache memories in all the Cm's - thus avoiding the need of replication and allowing the program to reside anywhere in the cluster, or even anywhere in the system. The program will eventually migrate to the cache of the appropriate processor that use it (i.e. partial replication will be automatically made in the cache).

The cache speed, compared to the primary memory speed, is not the main factor in the performance. Cache rate compatible with the local memory rate seems to be adequate in terms of cost, performance and complexity, and will be assumed in this study.

The size of the cache and the cache organization, for PDP 11 type architecture, was studied by Strecker [Strecker 76] and Mudge [Mudge 77]. The reader is referred to their work for general discussion on cache and its main parameters.

Strecker used an address tracing simulator, along with several test programs, to determine the parameters of the cache. The main results are given in the following table below (from [Mudge 77]).

Cache size (words)	Set size	Cache hit ratio (reads)
256	1	0.7
512	1	0.75
512	2	0.82
1024	1	0.87
1024	2	0.93
2048	2	0.93

From the simulation results the cache size for the PDP 11/60 was chosen to be of 1024 words, set size of 1, block size (number of words transferred from main memory) of 1, cache organization of "set associative" (and not "fully associative" which is still too expensive), and an anticipated cache read hit ratio of 0.87.

These results are quite compatible with what we could assume based on the data we have from the several benchmark programs, that were executed on Cm*. These programs took about 1700 to 3100, 16 bits words for the code (including different utility routines) and variables which were local to the Cm's (own's in Bliss programming language notation), large portions of which are rarely used.

In order to estimate the performance of a Cm* system with different cache sizes, we have translated the cache read hit ratio into system hit ratio i.e. total percentage of references

that do not require intra or inter-cluster reference. We've assumed the existence of a mechanism to differentiate between code and data private to the Cm - which can migrate into the cache, and "global" data structures that may be updated by another Cm[1].

Some consideration should be given to the *write* problem. Assuming the use of "write through" (i.e. writing both to the cache and the primary memory) than the (about) 10% writes in the system can make a significant difference in performance. We have measured that the total references to the *stack* and local variables areas account for about 15% to 25% of all references (we'll assume an average of 20%) and they also account for large percentage of the total *writes* in the system. The possibility of assigning storage areas local to each Cm for this purpose, improves the system performance as is seen in the table below.

Some other factors were not considered here. They include the I/O and context swap in the Cm that require invalidation of the data in the cache. Another factor, that appears in complex applications that use large number of code segments, is the need for overlaying these segments in the local Cm memory. Overlay causes degradation in the performance. This problem may be overcome by suitably large cache and segment invalidation mechanism. This problem is connected with the known "small address space" problem and will not be discussed here.

From the table given by Mudge, and information above, the following table was constructed. It shows the percentage of *miss* ratio due to various parameters: Cache size and set size, *miss* ratio due to reads, *miss* ratio due to writes (with local buffer - owns - plus stack local to the Cm or remote - in another Cm's memory.), *miss* ratio due to "global" data (including the writes to the "global" data) and the sum of the *miss* ratio's - i.e. the total percentage of references which are external to the Cm-Cache unit.

[1] This, in fact, may be done quite easily by using a mechanism similar to the "window" (mapping) registers used now, with one bit per register (segment) to allow the use of the cache.

Cache size/ set size	read miss ratio	writes- owns+stack local in the Cm	remote	"Globals" miss ratio (including writes)	Total system miss ratio
256 / 1	22.5%	0%		5%	27.5%
256 / 1	26.1%		8%	5%	39.1%
256 / 1	21%	0%		10%	31%
256 / 1	25%		6.7%	10%	41.7%
512 / 1	18.8%	0%		5%	23.8%
512 / 1	21.7%		8%	5%	34.7%
512 / 1	17.5%	0%		10%	27.5%
512 / 1	20.8%		6.7%	10%	37.5%
1024 / 1	9.8%	0%		5%	14.8%
1024 / 1	11.3%		8%	5%	24.3%
1024 / 1	9.1%	0%		10%	19.1%
1024 / 1	10.8%		6.7%	10%	27.5%
1024 / 2 or 2048 / 1	5.3%	0%		5%	10.3%
1024 / 2 or 2048 / 1	6.1%		8%	5%	19.1%
1024 / 2 or 2048 / 1	4.9%	0%		10%	14.9%
1024 / 2 or 2048 / 1	5.8%		6.7%	10%	22.5%

The results of the table are shown in Fig. 3.40 and the best results (with cache size of 2048 words and local variables local to the Cm) are plotted against the current system configuration results in Fig. 3.41.

From these results we can conclude that a cache of size 2048 words is a possible alternative to current configuration, trading off some performance for memory space (which is saved by avoiding code replications), for ease of storage management and time savings (by avoiding the need to load each Cm with it's local code), and by avoiding the need for overlaying code segments in the Cm's local memory as mentioned before.

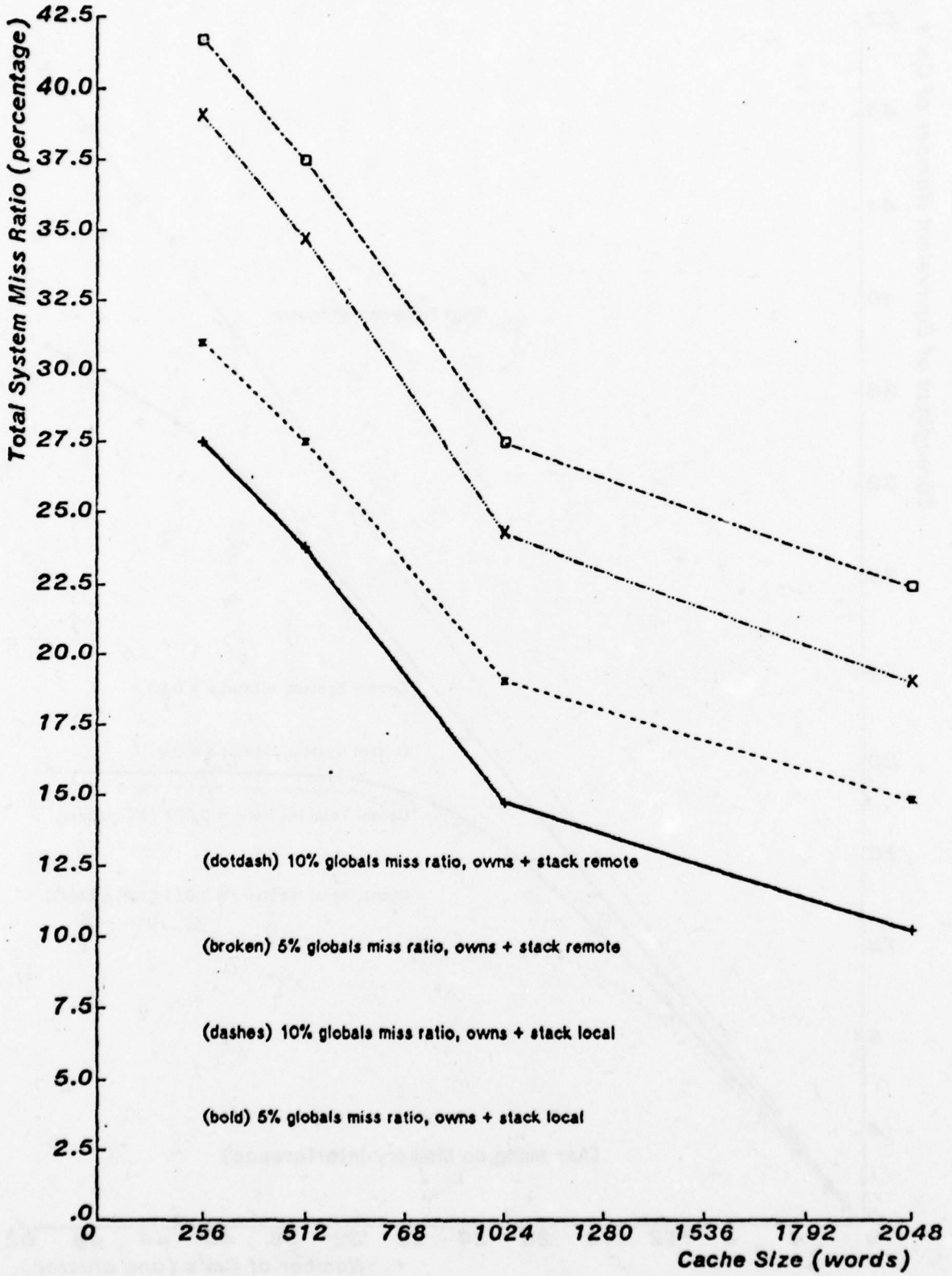


Fig. 3.40: Effect of Cache Size on Total System Miss Ratio

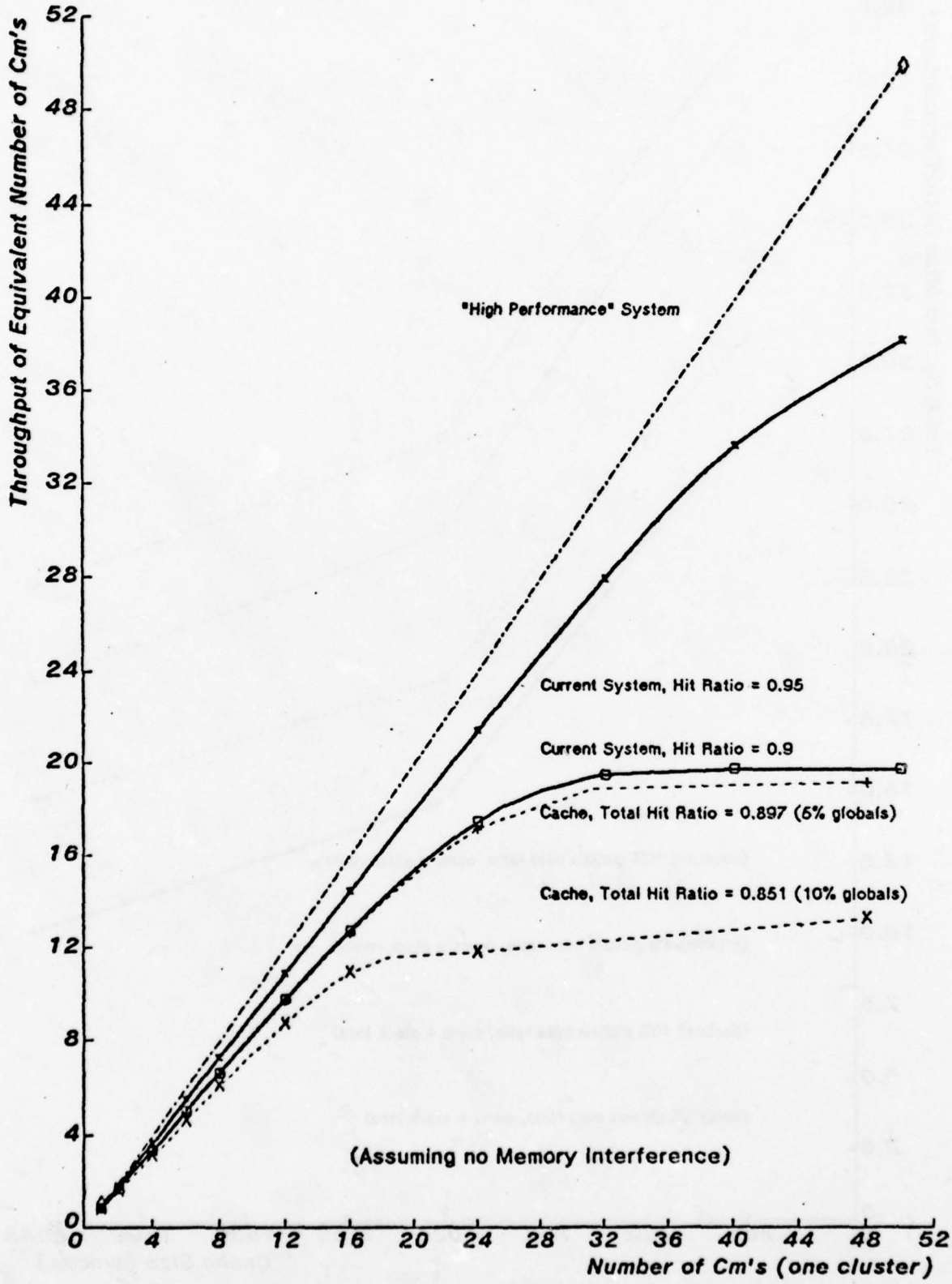


Fig. 3.41: Comparing Current System to System with 2K Cache

3.4.5 Input/Output Performance Aspects

In this sub-section some input/output aspects of the architecture were investigated. There is no input/output bound application on Cm* and thus no measured results. The following model estimates the I/O traffic impact on performance for some simple configurations. Two aspects were considered: The degradation in system's performance in the presence of I/O traffic, and changes in I/O traffic rate for various changes in system's configurations and parameters.

3.4.5.1 The I/O queueing network model

Fig. 3.42 show the queueing network model used to investigate the I/O questions. This model is similar to the multi-cluster network model with the inclusion of node 28 in cluster 1 to represent the I/O source, and of global class 6 to represent the I/O traffic flow in the network. Note that global class 6 flow is similar to global class 1 flow (in figure 3.3) with node 28 - I/O delay - replacing node 1 - CPU delay.

In this model three new parameters were added: MU28 - I/O inter reference delay, ALPHIO - ratio of I/O reference rate to a shared memory module to the total I/O reference rate, and BETAIO - ratio of inter-cluster I/O reference rate to total I/O reference rate.

We've used the model to investigate the I/O effect on the system's performance - we assumed the system to perform some application task in parallel with the I/O traffic and calculated the system performance with and without the I/O node, and the I/O rate was found by calculating the throughput of node 28.

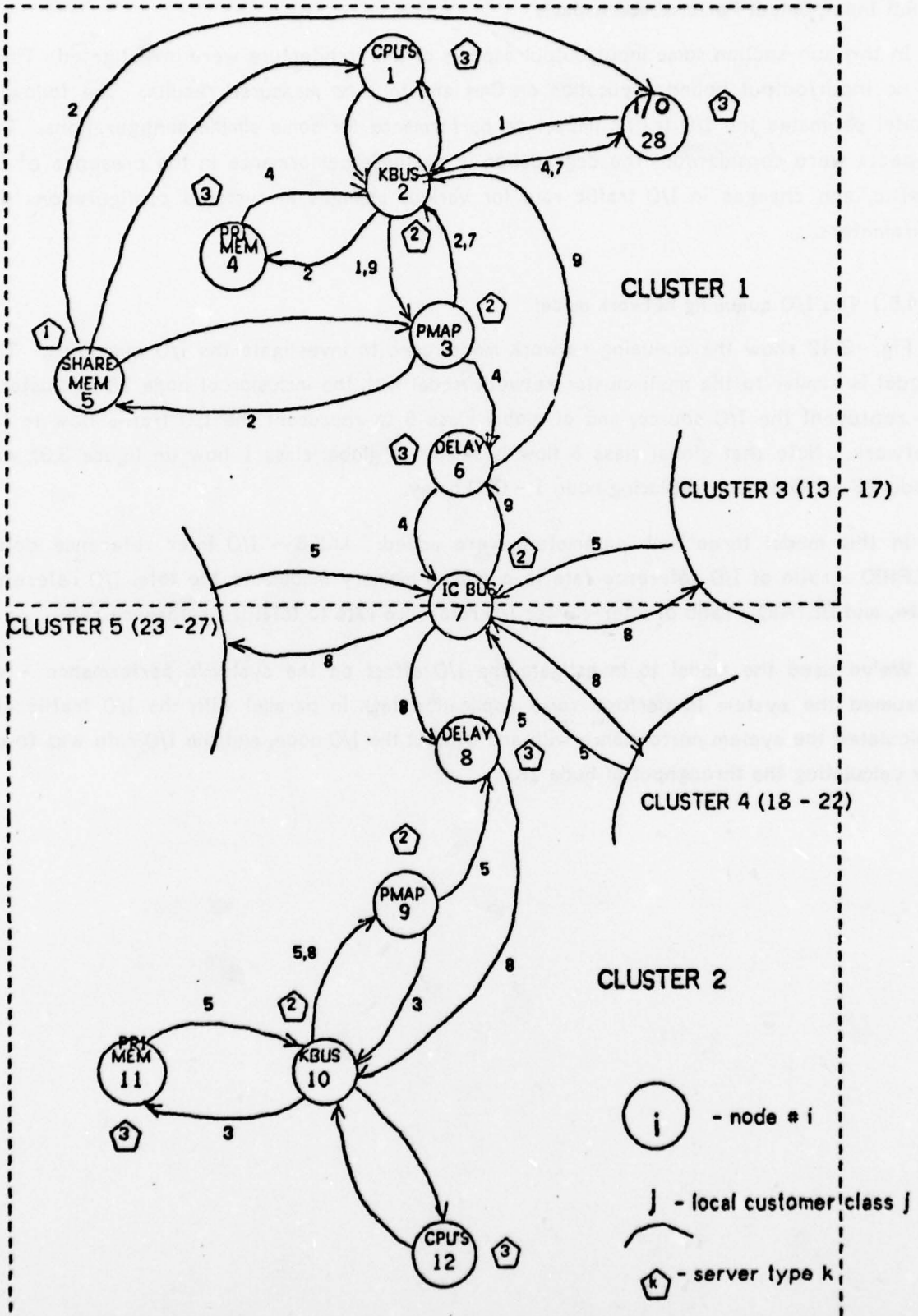


Fig. 3.42: Network Model - 5 Cluster Cmo System with i/o
Global Class 6 (flow from i/o)

clustio.spx

3.4.5.2 I/O performance results

The two I/O inter-reference delays (μ28) used in the calculation of the I/O network were:

- 13 μsec : representing a processor executing a tight loop:

```
LOOP: MOV (R1)+, (R2)+
      SOB R0, LOOP
```

Transferring blocks of data (which were read from an I/O channel to a Cm's memory) from one Cm memory to another.

- 0.5 μsec : representing a possible, sophisticated, DMA channel which is connected directly to the map bus and transfer I/O blocks to the Cm's memories.

One Cluster System

In Fig. 3.43 the effect of I/O traffic on the system's performance is shown for both of the above cases and application *hit ratios* of 90% and 95% (ALPHIO and BETAIO = 0). We assumed one Cm (or one DMA channel) engaged in I/O block, with an application executed in parallel. As expected the 0.5 μsec loop demand more time from the central (Map) resource and cause more performance degradation. The overall performance degradation is relatively small - maximum of about 8% (0.5 μsec loop and 25 - 30 Cm's in the cluster). The I/O node has an effect of just adding one more Cm with low local hit ratio.

Fig. 3.44 show the system's contention influence on the I/O traffic rate. We see that with low system contention the maximum I/O rate of the DMA channel (0.5 μsec loop) is about 3 times higher than the processor's (13 μsec I/O loop) I/O rate. The DMA maximum rate is about 150,000 words/sec. (75, 2K pages), and about 50,000 words/sec. (25 pages) can be transferred by the processor (neglecting memory contention and I/O transfer overhead). Higher degradation is experienced with lower system's *hit ratio* and increased number of Cm's in the system.

These results are for one I/O channel only (DMA or processor loop). It is possible to use a few I/O traffic block moves in parallel, e.g. in the Image Understanding preliminary evaluation [Chansler 77] it was proposed that an I/O channel (from a Disk) will be connected to two Cm's and transfers will be done from a few other Cm's in parallel. Memory contention becomes a problem in that case, for example with four Cm's executing I/O transfers in parallel (using the processor loop and transferring I/O blocks to their local memory), the maximum I/O rate was found to be only about 175,000 words per second due to a shared memory contention in the Cm which is connected to the I/O channel.

Note: Another (and better) way to implement the I/O transfer is to incorporate a block move mechanism. In this mode the Kmap will be responsible for the move of blocks of data from one Cm's memory to the other. Such a mechanism was implemented for the computer network emulation reported in the next chapter. Some changes in the queueing network I/O model would have allowed us to investigate this subject. Due to schedule constraints this was left for future investigations.

Two Cluster System

Fig. 3.45 and 3.46 show the results of incorporating inter-cluster I/O traffic in a two cluster Cm* system performing an application as described in Subsection 3.4.3 (50% of the application external references are inter-cluster). All the I/O traffic was assumed to be inter-cluster (BETAIO = 1) from one cluster (where the I/O channel reside) to the other cluster. Here, again, the inter-reference I/O delay parameters were assumed to be 0.5 μsec and 13 μsec, and all the inter-cluster transfers were of one word of data per transfer (and not the possible five to seven words per transfer that theoretically exist in an inter-cluster transfer).

The two bold curves in Fig. 3.45 are identical to those in figures 3.33 and 3.34 (two clusters without I/O), while the 4 other curves show the degradation in the system's performance in the presence of I/O traffic. The degradation is even smaller than in the one cluster case (maximum of 3% to 4%).

Fig. 3.46 show the inter-cluster I/O rate. 20,000 - 30,000 words/sec. (10 - 15 pages/sec. - depending on the system's parameters) might be expected in the actual Cm* system using the processor loop, and between 25,000 to 45,000 words/sec. - if a DMA channel was used. The results show that it is still possible to transfer I/O data blocks from one cluster to the other with I/O rate of about factor of 2 slower (for the processor loop transfer) or factor of 3 slower (DMA) than in the one cluster case.

Fig. 3.47 show the I/O rate of a few I/O channels executing the I/O block move. Here the I/O rate of one I/O channel (same as the dashed curves in Fig. 3.46) is compared to four I/O channels. The inter-cluster saturation effect can be seen when the four channel's I/O rate is less than four times higher than the one I/O channel rate. The degradation effect of the application (which is executed in parallel) is, again, significant. Similar curves can be done for other configurations.

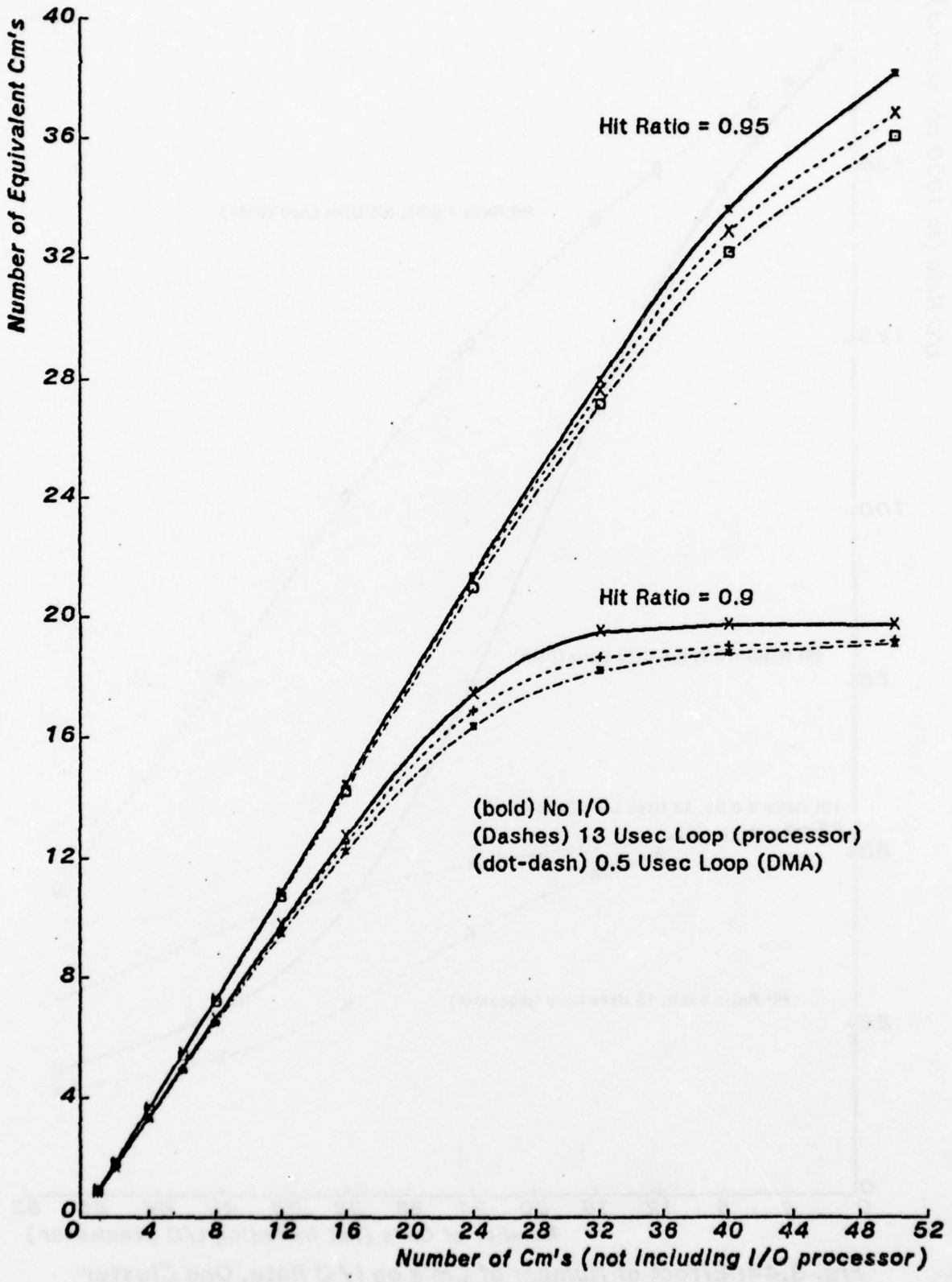


Fig. 3.43: Effect of I/O Traffic on Performance, One Cluster

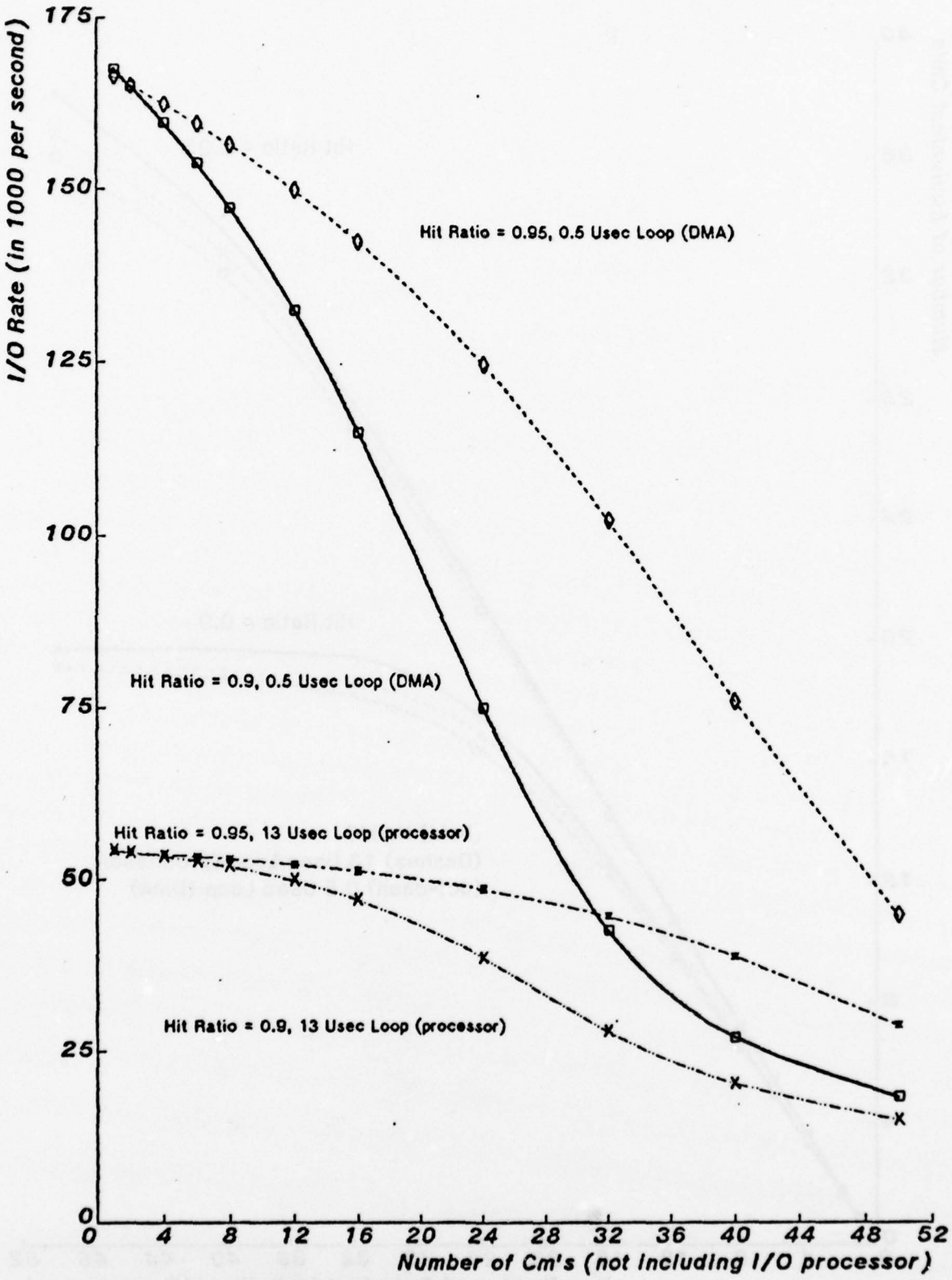


Fig. 3.44: Effect of Number of Cm's on I/O Rate, One Cluster

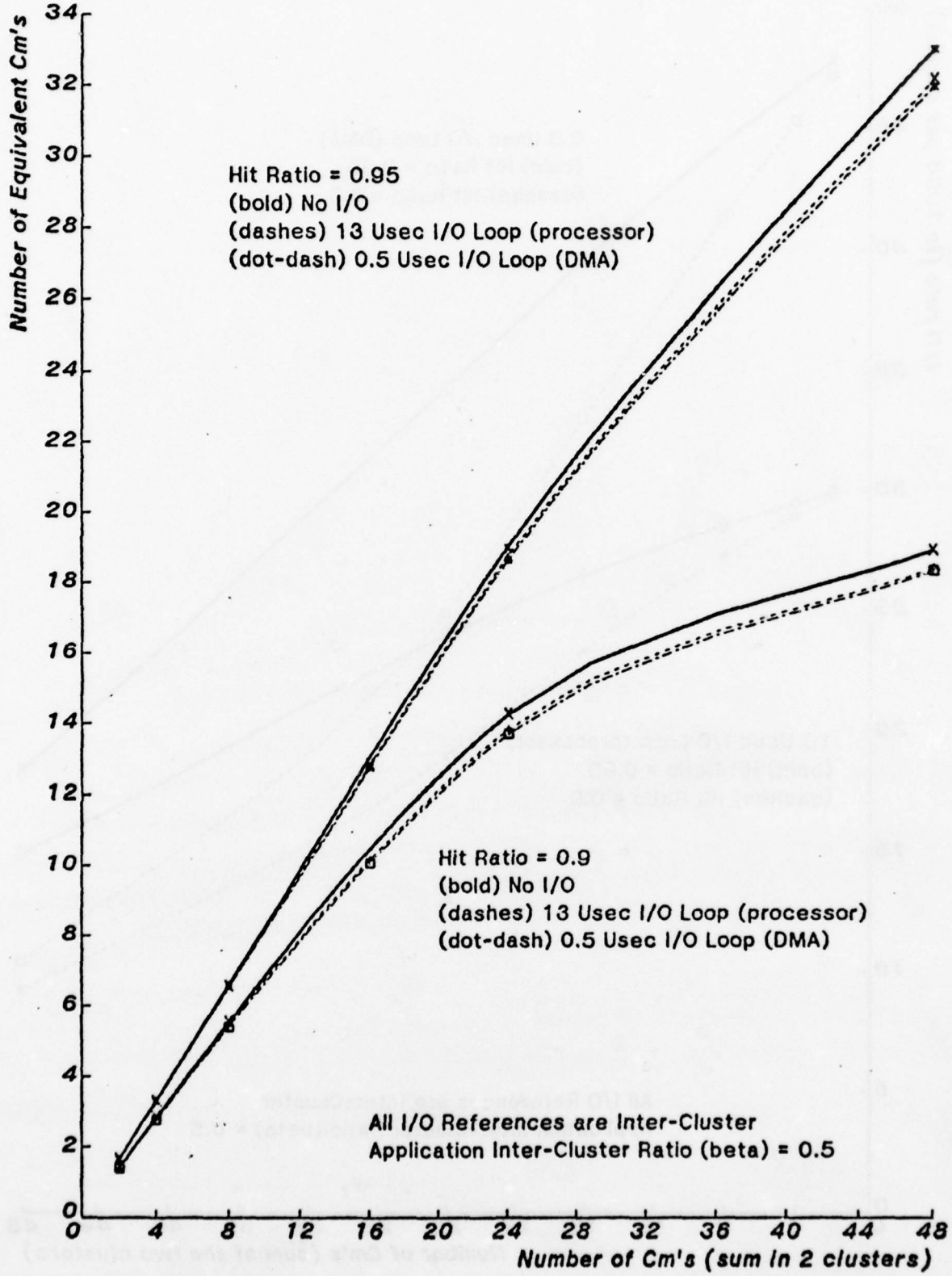


Fig. 3.45: Effect of I/O Traffic on Performance of 2 Clusters

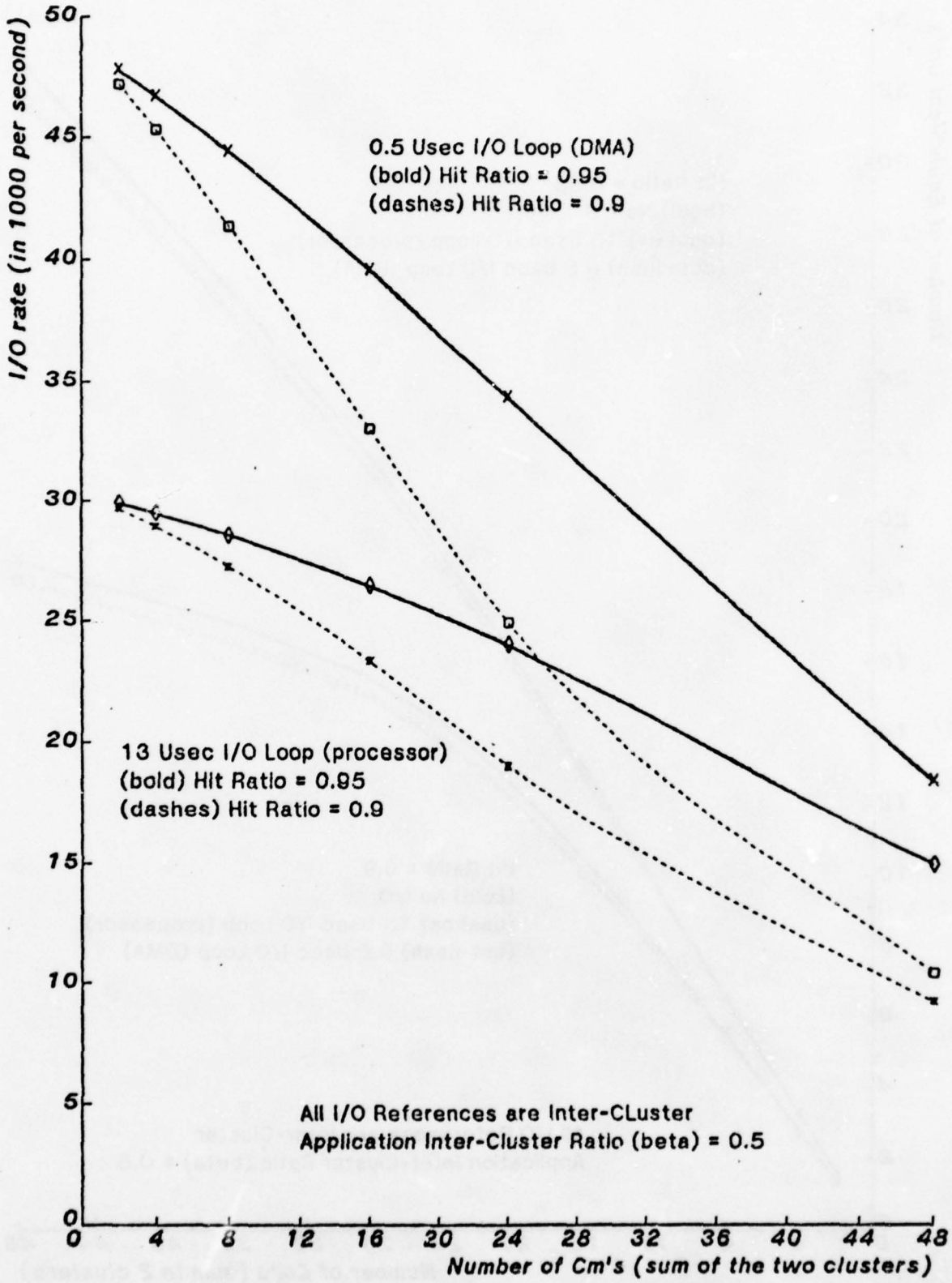


Fig. 3.46: Effect of Number of Cm's on I/O Rate, 2 Clusters

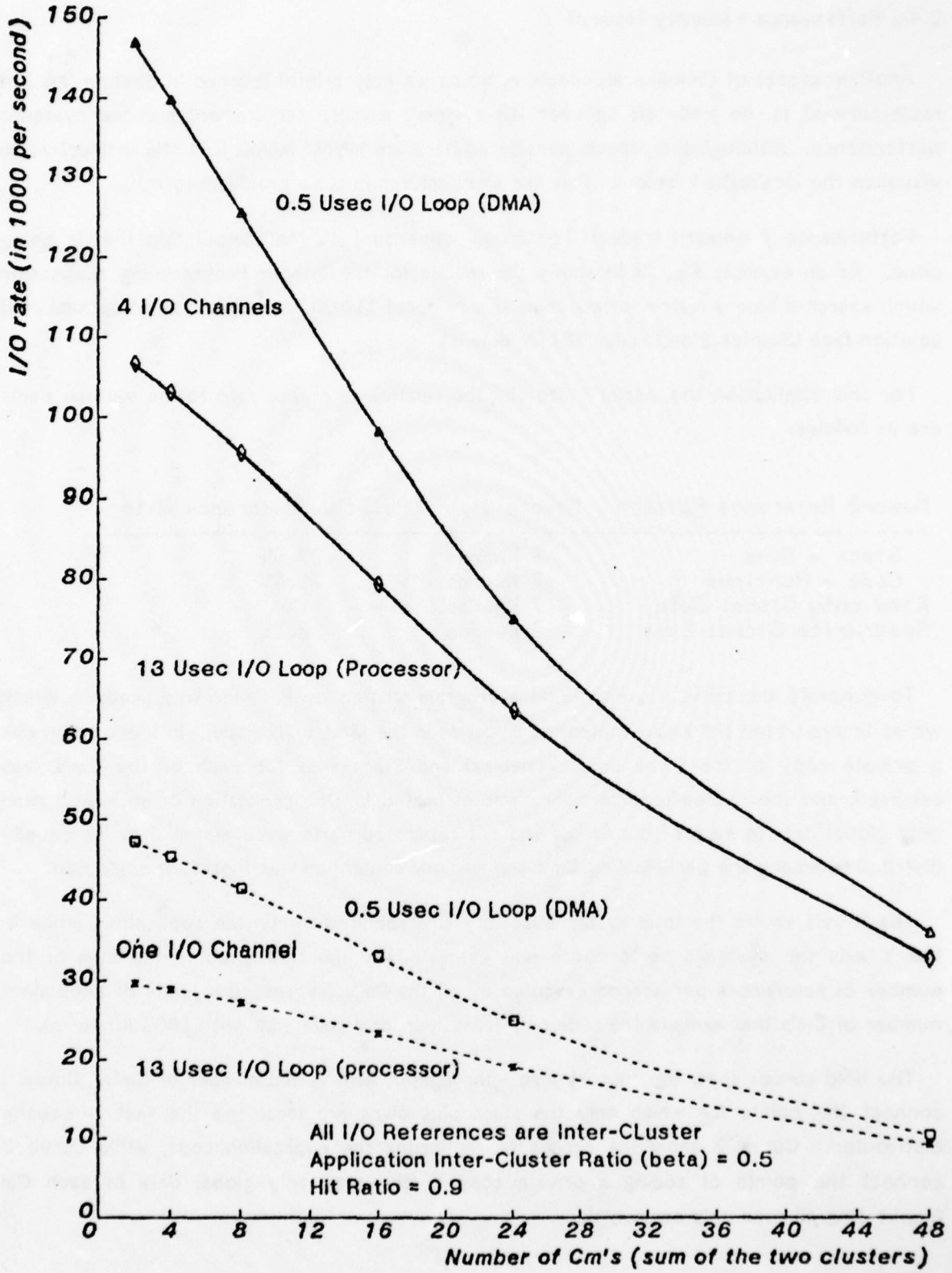


Fig. 3.47: Effect of Number of Cm's on I/O Rate, 2 Clusters

3.4.6 Performance - Memory Tradeoff

Another aspect of Cm*-like architecture, which we only briefly touched in Section 3.4 (*hit ratio* curves), is the trade-off between the system's memory requirement and the system's performance. Although the aspects considered here are almost trivial, it is still instructive to visualize the localization factor of Cm* like architectures in some graphical form.

Performance / memory tradeoff is strongly dependent on the computation that is being done. As an example Fig. 3.48 shows the results for the Integer Programming application which search a binary matrix (of maximum size of about 110,000 elements) for an optimal cost solution (see Chapter 2 and Fuller 78] for details).

For this application the memory size and the relative reference rate to the various parts are as follows:

Memory Reference Pattern	Memory Size	Relative Reference Rate
Stack + Owns	8 Kwords	27.5%
Code + Routines	2 Kwords	71.4%
Read only Global Data	7 Kwords	1%
Read/Write Global Data	2 Kwords	0.1%

To generate the curves in Fig. 3.48 a program written by P. Feiler was used, in which we've incorporated the Kmap contention model as in the M/M/1//N model. In these programs a private copy of the Owns (local variables) and Stack area for each of the Cm's was assumed, and the system's performance was estimated for the replication of code and read only global data in each of the Cm's. The not replicated parts were assumed to be equally distributed among the participating Cm's and memory contention was therefore neglected.

The X axis shows the total system's memory size required to run the application, while in the Y axis the system's performance was estimated by the throughput - the sum of the number of references per second executed by all the Cm's (expressed in units of equivalent number of Cm's that execute the code only from their local code - i.e. with 100% *hit ratio*).

The bold curves show the memory size - throughput with varied number of Cm's. Curve 1 connect the points for which only the stack plus owns are local and the rest is equally distributed. Curve 2 show the results of replicating the application code, while curve 3 connect the points of adding a private copy of the read only global data in each Cm (replicating all read only segments).

The results show clearly the significance of localizing code. Curve 1 show the poor performance of not localizing code (equivalent of only 2 to 3 local Cm's for 6 or more actual Cm's in the system). The addition of private code copies caused a steep increase in the slope of the bold curves (from curve 1 to curve 2, e.g. from point A to point B for the 6 Cms cluster) - achieving a significant performance increase with very small memory increase. The third part of the bold curves - from curve 2 to curve 3 (e.g. from point B to point C in the 6 Cms cluster) - is almost horizontal. This shows that replicating read only global data is useless - causing increased memory size with negligible performance gain.

Similar curves can be generated for other applications. The parameters needed can be either estimated or measured in a single processor version of the application (as was done in the Integer Programming case), and the best memory - performance application configuration can be chosen. Although the application used here is somewhat extreme (only 1.1% global data reference ratio) it still show the main points in the memory size / performance tradeoff - small increases in memory size for the most frequent used portions of the application program return significant performance gain, while global data replication give, in most cases, insignificant performance increase.

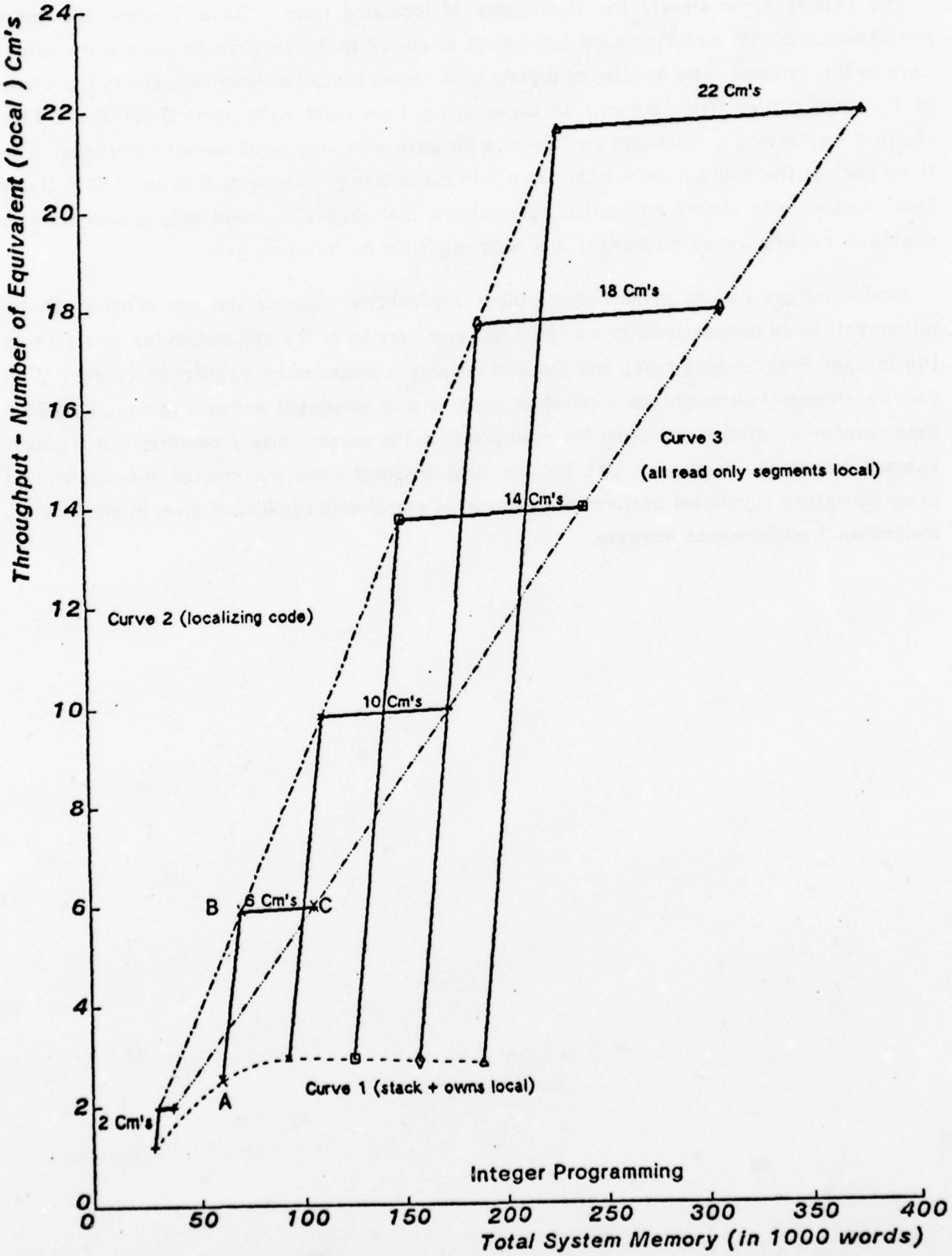


Fig. 3.48: THROUGHPUT AND MEMORY SIZE TRADE-OFF

3.5 Conclusions

Several performance models and their application to the evaluation of Cm* like multiprocessor architecture, were reported in this chapter. The practical methodology used can be similarly applied to the performance evaluation of any other large computer system.

The chapter presented the evolution of different models, with an emphasis on the queueing network analytic model. The detailed experiments that were conducted showed the practicality and provided an important validation to this class of models which which was rarely done before (and not for such a detailed model). Presentation and proofs (for the original work) of the various algorithms used in the models were given in the appendices.

Using these models we have investigated many interesting performance issues, which include exploring the performance of a larger system and various modifications and alternatives, some of which are now being considered. In particular we checked the performance of a system with faster components, compared one cluster and multi-cluster structures, investigated the possibility of using cache memories, the impact of input/output traffic on the system's performance, and performance - memory size trade offs. We commented on the implications of all these subjects in the appropriate sections. In Chapter 5 we have tabulated, summarized, and discussed the main results of this chapter.

The detailed results and many graphs included in the chapter give the designer of a new or modified, Cm* like, architecture a quantitative data on the performance implications of various aspects of the architecture, and hopefully provide any multiprocessor system's architect with some insight to the performance issues of such systems.

4. Performance of Local Computer Networks

4.1 Introduction

In Chapter 1 we introduced the notion of the distributed processing spectrum and the local computer network as part of that spectrum between the multiprocessor in one side and the loosely coupled, and geographically distributed, computer network - on the other side.

There is an increasing research and grows of local networking in the last few years. The list of existing local computer networks include the "Ethernet" at Xerox PARC [Metcalfe 76], the Distributed Computer System (DCS) at the University of California, Irvine [Farber 75], "Spider" at Bell Labs. [Fraser 75], DCLN at Ohio State University [Liu 75], HXDP at Honeywell [Jensen 78] and many others. Their main attributes are physical distribution of no more than 1000 meters, up to 256 processing elements and communication rates of 1 to 3 Mega Bits Per Second. The grows of the local networking use has brought standardization in the transport mechanism and hardware protocols employed.

Local computer network research issues range from hardware topology and protocols to network operating system and application programs decomposition and assignments. For surveys and discussions of the main research issues see [Kimbelton 75], [Enslow 77], [Gonzales 78], [Eckhouse 78]. Our interest lies in the performance aspects of such a structure[1]. Scanning the literature one finds that performance of local computer networks is one of the major issues, but there are no publications (to our best knowledge) on implementing an application program on a local computer network for performance gain, or on the requirements and performance issues involved. Performance investigation were made and models were developed for the design of the network transport topology (loop, ring, star etc.) and hardware protocol mechanisms (e.g in [Kleinrock 76], [Liu 77]) but they are of little use for the investigation of the performance of an application program utilizing such a structure to achieve performance in a manner similar to the way a multiprocessor is used.

As part of the performance investigation in multiple processor systems we wanted to show that a local computer network structure is capable of achieving performance gain for certain classes of applications. Although an accurate and objective hardware cost comparisons between multiprocessors and local computer networks are difficult to make and software issues and ease of programming complicates it even further, it seems intuitively true that a local network should cost less than a multiprocessor with the same number of processor - memory pairs. This is a result of simpler interconnections in the network and the increased

---

[1] Local networks have been used effectively for reasons other than performance. To connect *dedicated function* processors and terminals (e.g. a front end processor that controls communication with a main frame) and multiprogramming with resource sharing situations. Such cases are not considered here.

role of interconnection costs in the total system cost, as a result of reduced integrated circuit cost. Thus, the question of the more cost effective structure for the application becomes more valid. We tackle the performance comparison problem with the aid of hardware network emulation, experiments with application programs, and performance models.

We have used Cm* to emulate the hardware of a canonical local computer network, that we call Cm-Net. The objectives of the experiments with application programs were to enhance the performance of the structure. Thus, the requirements for fast response suggested implementing an efficient *process interface software support* package, with a low overhead, as the only level of abstraction between the application program and the hardware structure - eliminating the one or more operating system levels which are usually added to make the user interaction with the structure easier, as shown in figure 4.1. One can envision that an efficient, performance oriented, network operating system can be devised that will enable the user to utilize the network structure efficiently without incurring excessive overhead, and still allow him to implement his application in a convenient environment.

The chapter starts with a description of the way Cm-Net was emulated and a functional description of the *process interface software support* package. A short description of the two applications that were implemented and the decomposition issues involved is discussed next. The detailed experiments and measurements that were conducted are then presented. A simple queueing network performance model was developed and its results were compared to the measurement results and its possibilities and limitations are discussed. The model was used to investigate a few performance issues.

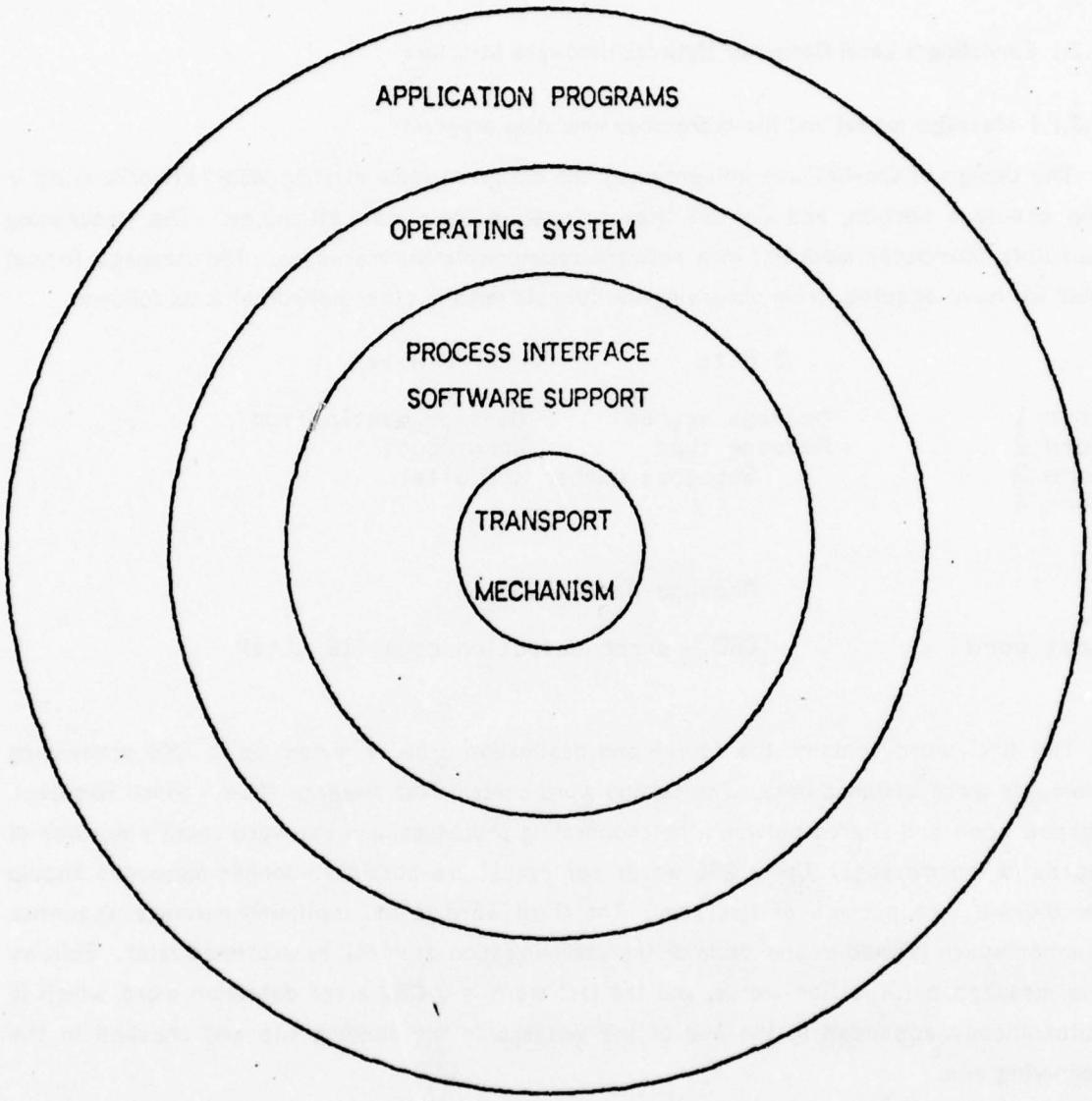


Fig 4.1 Levels of Abstraction

levels.spx/b

4.2 Description of Cm-Net

4.2.1 Emulating a Local Computer Network Hardware Structure

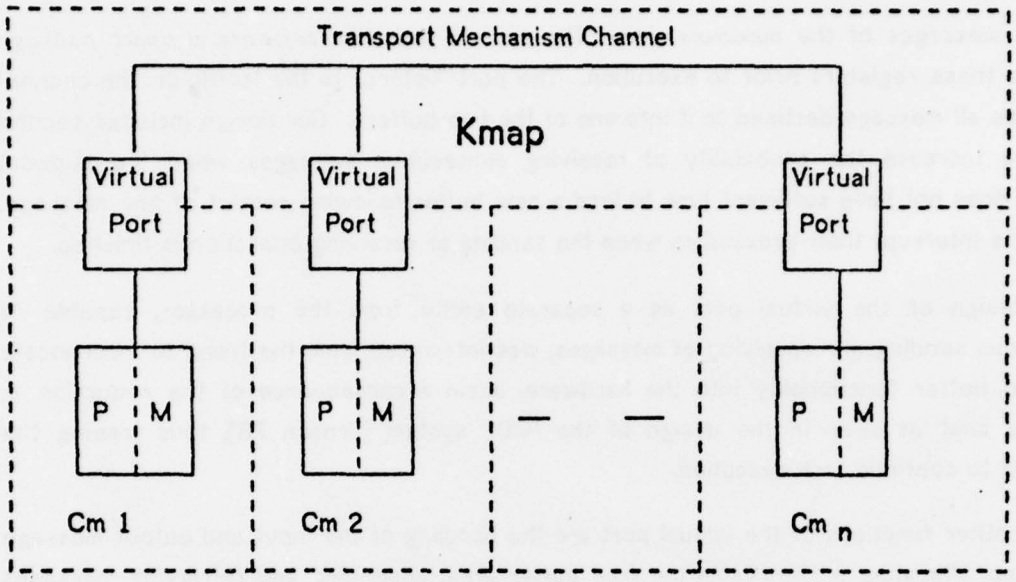
4.2.1.1 Message format and the micro-code emulation program

The design of Cm-Net was influenced by the design of some existing local networks cited in the previous section, and Cm-Net tries to capture their main attributes. The processing elements (computer modules) in a network communicate via messages. The message format that we have adopted (from observing the formats used in other networks) is as follows:

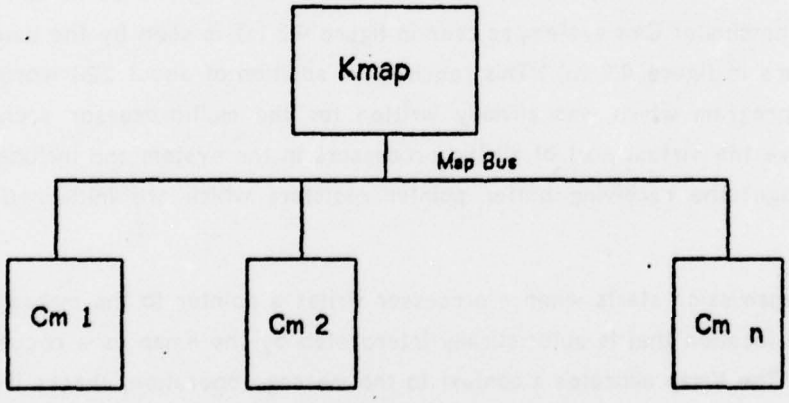
	8 Bits	8 Bits
Word 1	Message source	Message destination
Word 2	Message type	Word count
Word 3	Sequence number (16 bits)	
Word 4	.	
.	.	
.	.	
.	Message data (16 bits)	
.	.	
Last word	CRC - error detection code (16 bits)	

The first word contains the source and destination process names; up to 256 processing elements were assumed here. The second word contains the message type - which has been agreed upon and shared between the cooperating processes, and the word count - number of words in the message. Up to 256 words per packet are possible - longer messages should be divided into packets of this size. The third word is the (optional) message sequence number which is used in one mode of the communication protocol, as explained later. Follows the message data portion words, and the last word is a CRC error detection word which is automatically appended to the end of the message in the sending site and checked in the receiving site.

Figure 4.2 shows a PMS level diagram of both the actual and virtual hardware architecture that we have used for the network implementation. The virtual network works as follows: A process in a processor P constructs a message in its private memory M. A pointer to the message is given to the virtual port which retrieves the message from memory and sends the required number of words out into the transport mechanism channel when the latter is available. A simple FCFS discipline was assumed here. The port at the destination process has two hardware registers as pointers to two buffers in its memory, capable of storing



b. The Virtual Hardware Architecture



a. The Actual Hardware Architecture

Fig. 4.2 Cm - Net, Cm* as a Local Computer Network

netwrk.spx/b

incoming messages of the maximum size. The *process interface software support package* initializes these registers prior to execution. The port "listens" to the traffic on the channel and copies all message destined to it into one of the two buffers. Our design includes second buffer to increase the probability of receiving consecutive messages when the support program does not have sufficient time to load a new buffer following receipt of one message. Both ports interrupt their processors when the sending or receiving operation is finished.

The design of the virtual port as a separate entity from the processor, capable of autonomous sending and receiving of messages, was influenced from the trend to incorporate more and better functionality into the hardware, again a consequence of the reduction in hardware cost as seen in the design of the HXDP system [Jensen 78], thus freeing the processor to continue task execution.

Three other functions of the virtual port are the handling of the input and output message queues (i.e. allocating buffers from the free buffer area, appending and retrieving messages from queues etc.), the automatic handling of acknowledgements, and the (not implemented) generation and checking of the CRC word.

In the actual Cm* system most of the port functions and the serial line transport channel were emulated in the Kmap. As described in chapter 1 the Kmap is a special purpose, 80 bits wide microprogrammed controlled, processor. We have microprogrammed it so that the architecture of a one cluster Cm* system, as seen in figure 4.2 (a), is seen by the user as the network architecture in figure 4.2 (b). This required an addition of about 220 words to the initialization microprogram which was already written for the multiprocessor architecture. The Kmap serves as the virtual port of all the processors in the system and includes (in its internal data storage) the receiving buffer pointer registers which are initialized by the processors.

The message transmission starts when a processor writes a pointer to the message to be sent into a special location that is automatically interpreted by the Kmap as a request for a message transfer. The Kmap allocates a context to the message operation, checks if another transfer is not in progress and store the request in an internal FCFS queue if another transfer has already started, frees the processor to continue operation, reads the first two words (the message header) and check if a buffer is available in one of the two buffer pointer registers of the destination processor. The *context* in the Kmap then transfers "word count" words from the buffer in the source processor's memory into the destination processor's memory. The Kmap interrupts both the source and destination processors to inform them on the completion of the message transfer, and then frees the *context* - ready for a new message transfer.

If a buffer is not available in the destination site the Kmap wastes the same amount of time, as if a message was in fact transferred in the bus, by reading the word from the source message memory and writing it back into the same location. The Kmap interrupts only the source processor and sets a special flag in the source message first word to indicate that case. The Kmap allows only one message transfer at any given time, these last two features simulate the behavior of a real network.

Other features of the micro-code include the possibilities of setting a delay function between successive word transfers, thus allowing simulation of variable I/O rates, and a statistics micro-package that counts the number of messages and words that were transmitted. These last features are an advantage of the emulation, as such functions are difficult to achieve in a real network.

To simplify implementation the buffer allocation, queues handling and acknowledge mechanisms were done by the *process interface software support* package, i.e. part of the virtual port function were done at the expense of the processor time.

4.2.1.2 Timing measurements

Timing measurements were conducted to determine the time it takes for a message transfer. It was found that the shortest packet possible (3 words) takes about 85 μsec and each additional word takes about 12 μsec (this was done in a period when the Kmap clock was slowed down to 180 nsec, somewhat faster times should be expected for normal clock of about 158 nsec). It was determined that this overhead consists of about 18 μsec delay to execute the (about) 100 micro-instructions in the Pmap necessary to complete one packet transfer, 26 μsec delay in the processor interrupt and move instruction to restart a new transfer, and about 41 μsec in Kbus operations and DMA accesses (9 Kbus and DMA operations). For the word transfer about 9 μsec is spent in Kbus and memory reference delays, and 3 μsec in micro-instructions in the Pmap. These measurements show that the maximum possible I/O rate of the emulated network is about 1.3 MBits/second.

4.2.2 The process Interface Software Support Package

4.2.2.1 The functions

The process interface software support package is a collection of application independent routines and was intended to supply the necessary functions that the application program might request, in an efficient manner. The functions performed by the routines include:

- Initialization - to construct a free buffer linked list and to detect all the available

processors in the network willing to participate in the task.

- Interrupt - to handle sending and receipt of messages.

- Linked list manipulation - to append, delete and retrieve messages from the three doubly linked lists (where the received messages, messages to be transmitted and messages awaiting acknowledgements are being kept) in the correct order that they were generated.

- Process interface - to allocate a buffer to a process, to enable a process to send a message to a particular process or broadcast a message to all the participating processes.

- Timing and statistics - collection and printing.

- Acknowledge mechanisms (to be explained later).

- Message decoding - to decode the incoming message type and call the appropriate process routine that will handle the request.

This collection of routines and data structures requires about 1400 words. Buffer area large enough for 18 packets (4600 words) was used.

4.2.2.2 The acknowledge mechanisms and protocol

We wanted to ensure reliable message transfer as well as to investigate the effect of acknowledge mechanisms on the local network's performance, and thus implemented two acknowledge options: **hardware** (automatic) acknowledge and **software support** acknowledge mechanisms.

The **hardware** acknowledge option follows the HXDP network where acknowledges and retransmissions are the responsibility of the port. In the Cm-Net case the reason for message loss is the lack of a buffer in the receiving site. This is detected by the Kmap which set a *flag* in the sending message first word. The process interface interrupt routine checks this bit and retransmits if the *flag* bit is set, or discards the message if not. A real system will require a more sophisticated hardware acknowledge mechanism to deal with other causes of message loss.

The **software support** acknowledge mechanism option is more elaborate and worth special attention. Each message contains a unique sequence number and, in this mode, a specific acknowledge message is to be sent back for each message that was received correctly. Each processor assigns four memory locations for each (possible) other processor in the network that he might communicate with. These words, and their functions, are:

- "Last received" - Keep track of the last message received correctly from that

processor, thus insuring the receipt of messages in the correct order and discarding any erroneous or replicated message.

- "Next send" - The sequence number to assign to the next message to be sent to that specific processor.

- "Received OK" - The sequence number of the last acknowledged message.

- "Old sent" - A copy of "next send" after a predetermined time delay interval.

The mechanism works as follows: each message that was sent is appended to a doubly linked list to keep them in the order in which they were generated. For each acknowledge received with the sequence number higher than the "received OK" number the link is searched, and all messages with sequence number equal, or lower, than the one acknowledged are discarded. The "receive OK" word is next updated, this solves the problem of lost acknowledges.

Using the real time clock mechanism, in each time interval (we arbitrarily chosed one tenth of a second) the "old sent" sequence number is compared to all the messages awaiting acknowledges. All those with lower sequence number are retransmitted and the current "next send" sequence number is copied into "old sent" to be used in the next time interval. This mechanism ensures reliable communication protocol between all processors participating in the task.

4.2.2.3 Timing measurements

Measurements were made to determine the overhead encountered when using the process interface software support package. For these experiments we have changed the application program to transmit and/or receive messages in the highest rate possible and the statistics package supplied the necessary parameters. The measurement results were:

- Sending a short message from the user environment (includes buffer allocation, message generating and sending, and buffer releasing): about 1.1 msec./packet.

- Receiving a message (includes buffer reallocation , message decoding, and buffer releasing): about 0.7 msec./packet.

- Acknowledge (send or receive): about 0.6 msec/packet.

4.3 Application Programs and Decomposition Issues

4.3.1 Goals

Two application programs were chosen as candidates for the performance evaluation of the computer network. The goals that led to their choice were:

- To enable comparisons to the multiprocessor structure the applications should have been implemented and measured on Cm* configured as a multiprocessor.
- To show that local computer networks are capable of executing certain classes of application programs efficiently, the applications should possess an *a priori* promising properties (e.g. low communication demands).
- Somewhat contradictory to the previous goal we wanted to see what application programs executes poorly on local networks in comparison with multiprocessor execution.
- The application programs should differ enough to illustrate different performance issues and aspects of local networking.

The two application programs chosen were the Integer Programming application and the Harpy speech recognition system. Both were implemented and measured on Cm* as a multiprocessor and they seemed to be on the two extremes of the possible application candidates; Integer Programming with low communication overhead and Harpy with complex and intense communications demands.

In the problem decomposition the guideline was to try to first find the best decomposition of the algorithm for the given structure and then experiment with some variations in the decomposition.

4.3.2 Integer Programming Decomposition

A description of the Integer Programming application was given in chapter 2 and will not be repeated here. Exactly the same algorithm was used in the network version. We again used the *master - slave* relationship between the processes with the *master* as the one that initializes the task, schedule the work done by the *slaves* and print results. Recall that the *master* participates in the task as any other process.

Two versions of the Integer Programming application were experimented with. In the first version we tried to utilize the network structure to its full potential at the expense of memory utilization. As before the matrix was created by the *master* process but in the

beginning of the task execution the matrix, and the necessary *global* read only variables, were sent and replicated in the private memory of all the processors that were to participate in the task. The only communication during execution, later on, were requests from the *slaves* for new assignments (new initial search paths), the *master* supplied new assignments, and the *best cost* messages that were broadcasted from the process that found a new, better, solution (to allow all the processes to prune their future tree search with the best cost function available).

The second version illustrates the case where cost considerations limit the memory size of each of the processing elements in the network such that it is not sufficient to hold all the read only *global* data. For ease of implementation a heterogeneous distribution of memory size in the network was assumed with the *master* process having sufficient memory (and addressing capabilities) to hold and directly access all the data needed for the application. For this version we assumed that each *slave* process is capable of holding one vector of the matrix in any given time, and he must request a new vector from the *master* process whenever he needs it. The maximum matrix size was less than 8000 words for the Integer application - one bit for each matrix element, but similar (not binary) matrix problems would require excessive memory size if a copy in all the Cm's is required. This is, in fact, another example of the traditional cost/performance issue.

Many other decomposition alternatives that affect the performance are possible, including implementation of more than one *master* (where each is responsible for part of the matrix), equally dividing the matrix between all participating processors, etc. We felt that these two implemented versions will be sufficient to clearly illustrate the important issues.

4.3.3 Harpy, Speech Recognition System

4.3.3.1 The application

Harpy is an speech recognition system developed for a uniprocessor at CMU [Lowerre 76] and was later decomposed into a set of cooperating parallel processes and implemented on C.mmp (another multiprocessor at CMU) [Oleinick 78]. It was modified and implemented on Cm* as a multiprocessor by P. Feiler ([Feiler 77], [Jones 78]).

The Harpy task implemented on Cm* (that was also implemented on the computer network) is the desk calculator task (DESCAL) which has a vocabulary of 32 words. Speech sentences are of the form "Alpha gets four times gamma". The speech sentence (utterance), which takes 3 to 4 seconds, is digitized and preprocessed on a PDP-10 computer. The preprocessed data is the input data for the Harpy program on Cm* which attempt to recognize the sentence.

This input is composed of probability vectors that a segment of the utterance match a possible phonic gesture of speech. The algorithm relies on a predefined network in which nodes are phonetic gestures and directed links are probabilities that one node follows another. The recognition process is a heuristic parallel search through the network. The DESCAL network contains about 1000 nodes. The average branching factor in the network was about 4 and the number of node candidates for each segment of speech was measured to have a mean of nine, but a large variance (e.g. 30% of the segments has less than 4 candidates). The small number of candidates limits the efficiency of solving this application with many processes in parallel. For more details on the algorithm see the references cited above.

4.3.3.2 The multiprocessor and computer network decompositions

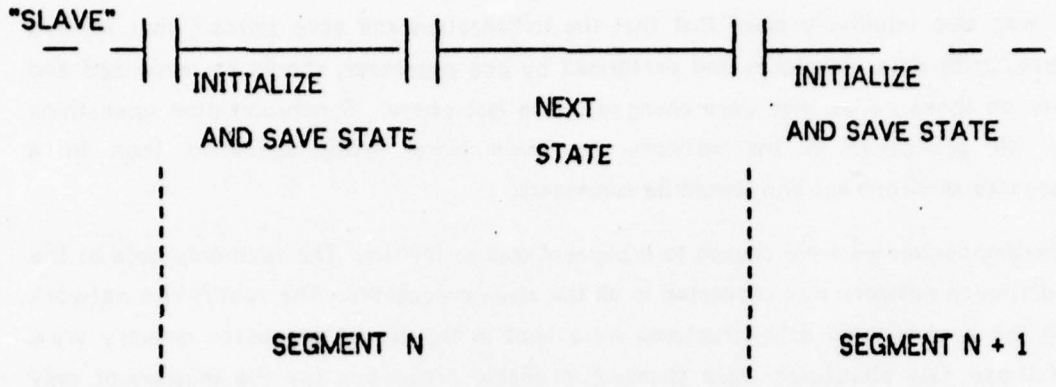
In the multiprocessor version the *master* process initializes the task and synchronizes the *slave* processors during the task execution. This is done in three steps for each segment of speech (about 30-50 segments per utterance): Check of all the possible next nodes of the current candidates in the network (next state), network initialization (initialization), and prune of the search results and save of the candidates for the next segment (save state).

The assumption that many processes can participate in the task led to various implementation decisions. The *master* does not participate in the actual execution of the task and thus was available for fast synchronization of all the *slave* processors in each step. This resulted the decomposition shown in figure 4.3 (a).

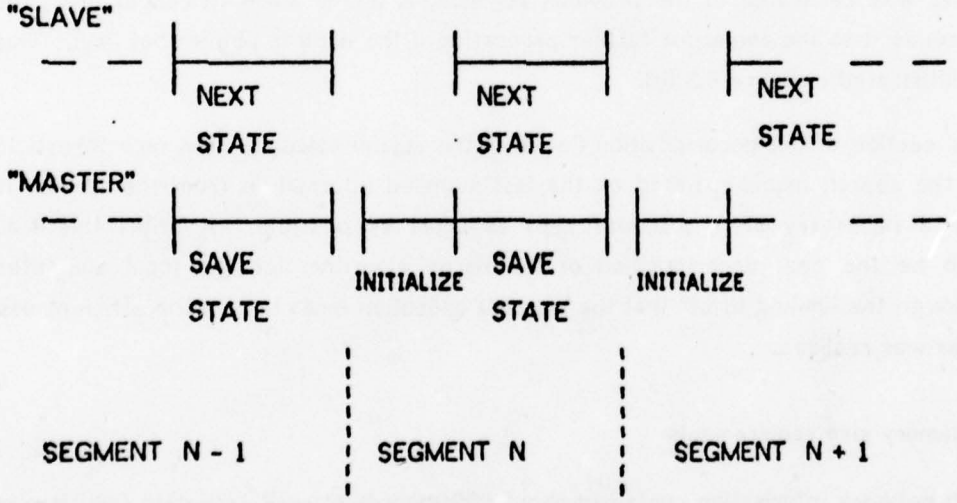
Other decisions involved the *global* read/write data structures, that mainly contain information about node probabilities and other node related information. Those data structures were organized according to the node's names and thus the initializations and saving steps involved initialization and searching by all the *slaves* of those (1000 elements) data structures looking for new information and saving the relevant ones. These decisions seems to lead to a good decomposition for a multiprocessors with many processors as the cost of accessing *global* data is relatively low.

For the computer network implementation of Harpy it was clear that the multiprocessor decomposition has an extremely high communication rate, unbearable in the network environment. To make a fair comparison between the two structures the algorithm should be decomposed differently in a way best suited for the structure[1]. The access rate to *global* data should be minimized and the read/write *global* data should be stored local to the process

[1]We make a distinction here between an algorithm - the decisions and heuristics used to solve the problem on a uniprocessor and the decomposition of that algorithm into parallel tasks. We tried to keep the algorithm intact but changed the decomposition for the network implementation.



a. HARPY'S *DECOMPOSITION ON A MULTIPROCESSOR*



b. HARPY'S DECOMPOSITION ON A COMPUTER NETWORK

Synchronization Points (by master)

FIG. 4.3 HARPY'S DECOMPOSITIONS ON Cm*

harpy3.spx

that need a direct access to it. Assembly and de-assembly of the messages by a process would require more time than the processing time done on that data in the remote process site. It was also intuitively clear that that the initialization and save phases, that involve those read/write data structures and performed by one processor, should be minimized and done only on those nodes that were changed in the last phase. Synchronization operations between all processes in the network are even more costly operation than in a multiprocessor environment and should be minimized.

The decomposition we have chosen to implement was as follows: The read only data of the searched speech network was replicated in all the *slave* processors. The read/write network data and the accompanied data structures were kept in the *master* processor memory area. Some of those data structures were changed to enable processing (by the *master*) of only those nodes that were changed, by the last segment, thus cutting the processing time of those phases considerably.

To reduce the synchronization overhead, and reduce the processing time even further, we have pipelined the execution of the *master's save state* portion of the algorithms with that of the *slave's next state*. The *master* checks if the probability of a node to become a new candidate, that was calculated in the previous segment, is above some threshold limit and immediately sends it to the *slaves* for further processing if the node is above that limit. This pipelining is illustrated in figure 4.3 (b).

The critical section of the decomposition (in which the *master* calculates the new threshold function for the search pruning, based on the last supplied information from the *slaves*) is unfortunately a necessary step, and was kept as short as possible. A priori this was considered to be the best decomposition of the Harpy algorithm for the local computer network, although the limiting effect that the *master's* execution times has on the efficient use of many *slaves* was realized.

4.3.3.3 The memory size requirements

The speech network information contained about 6300 words of *read only* data (which was replicated in all the *slaves*) and about 3000 words of *read/write* data (kept in the *master's* process area). The preprocessed speech sentence required about 2500 words for each sentence (the probability vectors). This is not needed for a live processing when the segments come in succession from the pre-processor.

The *master* code is less than 2000 words, about 8000 words are needed for the various *read/write* data structures manipulated by the *master*. The *slave* memory contains the 6300 words of the speech network *read only* data, 650 words were sufficient for the *slave's* code.

Memory space was also needed in each processor for the *process interface software support* package and about 1000 words for some I/O and library routines.

note: DESCAL is the smallest task of the four existing Harpy tasks. Considering the largest task (AIX05) it has a vocabulary of about 1000 words, 14400 nodes, 41500 pointers between the nodes in the network and a branching factor of 2.9 [Oleinick 78]. These numbers lead to a requirement of about 78000 words for the speech network and associated *read only* data. We will comment about the implications of network size differences in the next section.

4.4 Measurement Results of Cm-Net

4.4.1 The Integer Programming Application on Cm-Net

In this subsection the results of the measurements are presented by various figures. We mention here the main conclusions from these measurements and will return to these results for further discussion and explanations in connection with the performance model in section 4.5.

Figures 4.4 to 4.6 show the measurement results of comparing the execution times and speed up factors of the Integer Programming problem solved both on CM-Net and Cm-Multiprocessor. The network execution time contained the initialization time needed for replicating the matrix and the global read only data. The bold curves represent the execution time of the five cases of the Integer Programming and the dashed curves show the multiprocessor results. These results are extremely encouraging. They show that for some classes of applications with low communication overhead, the network overhead is still negligible even for a modest I/O rate of about 1.3 Mbits/second. Notice the more than linear speed up achieved in case 1 of figure 4.6.

Figure 4.7 shows that the acknowledge mechanism has also a negligible effect in that case. This results from the small number of messages (and therefore acknowledges) that are being transferred during execution.

The next two figures (4.8 and 4.9) show an example of the results collected by the statistics package - the total number of packets and words that were transferred during the execution of case 1. The results show that the number of packets grows rapidly from one to two processes, and continue to grow in a slower and almost linear rate later on. The addition of the second process causes many requests for new assignments (about half of the total number) to be sent between the *slave* and *master* processes. Adding more processes results a modest grows in the message rate for initialization of the new processor and as a result of the additional work that is taking from the master and done by the slaves. The number of packets for the *software support* acknowledge mechanism is about twice the number of the *hardware* mechanism, as expected.

The number of words in figure 4.9 grows almost linearily. This is explained by the task initialization phase. The initialization phase of replicating the matrix and the *read only global data* contains only five packets with (almost) the maximum number of words possible in each. The messages that are being sent during execution are short (5 to 6 words each) and the total word traffic size is dominated by the initialization phase. The short acknowledge

messages add only 10% to 20% to the total number of words sent (while duplicating the number of packets).

Figures 4.10 and 4.11 show the effect of the message transfer rate on the execution time and the speed-up factor for case 1. The results show that this application is insensitive to changes in the network I/O rate, and the rate could be reduced to about 50 Kbits/second without affecting the performance much. Only extremely slow network channel rate causes severe performance degradation (again a result of the low traffic demand). It is interesting to observe that with a slow link - 5 Kbits/second, and two processes the application program speed-up is still almost linear.

Figures 4.12 to 4.15 show the measurement results of the second version of the Integer Programming where the matrix is centralized and one vector at a time is transmitted to a *slave* upon request. These results show clearly the crucial and sensitive role of the performance/memory size tradeoff in the network environment. While cases 1 and 2 showed a severe degradation, but still positive speed up factor with up to 3 or 4 processors, case 4 showed a considerable slow down. This phenomena is explained by the relatively long execution time of a slave for each sparse vector received in those cases, while the vector in case 4 has much higher density and is immediately found to be contradicting to the solution, causing a short execution time in the *slave*, and high message rate. This, in turn, results in a considerable degradation in the master's useful execution time (its time is spent in assembling and sending of the required vectors of the matrix) - thus slowing down the total execution time.

Figures 4.14 and 4.15 show the measurement results of the total number of packets that were sent during execution of this second version of the Integer Programming application. These numbers were compared (the *dashed* curves) to the measurements of *twice* the number of nodes (the *bold* curves) that were searched by the processors. Each node requires two messages, one from the *slave* to the *master* and one from *master* to *slave*. These results show that with 5 to 6 *slave* processors in cases 1 and 2, the two curves intersect, i.e. we can't expect more performance speed-up gain by adding more processors (as is verified in figure 4.12). The number of packets in case 4 show that performance speed-up cannot be expected with more than three processors. Because the *master* saturates, as explained in the next section, performance speed-up was not achieved at all.

The memory size saved by centralizing the matrix in the second version is about 750 words/processor in cases 1 and 2 and about 4000 words/processor in case 4.

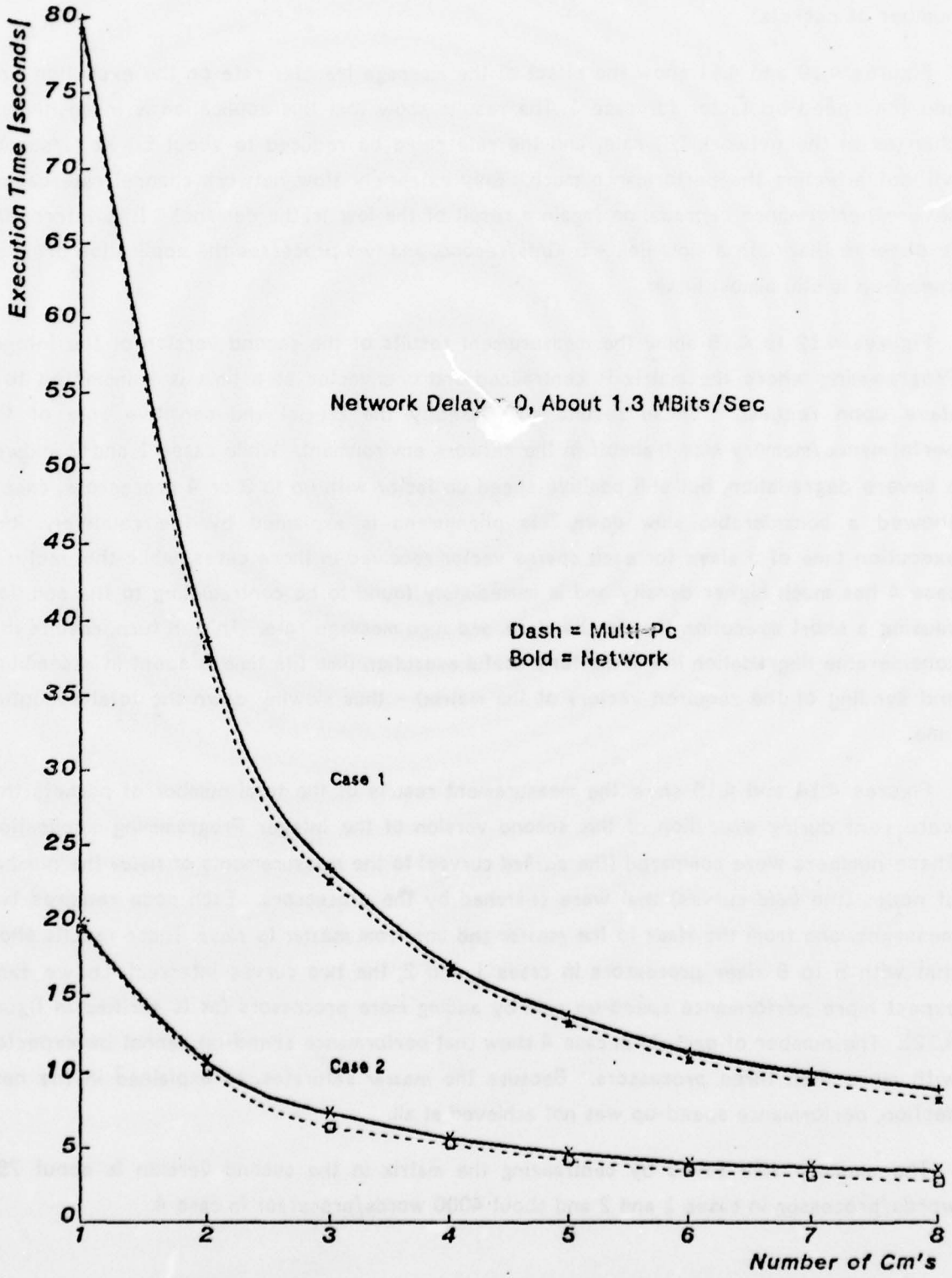


Fig. 4.4: Integer Programming, Comparing Network and Multi-Pc (a)

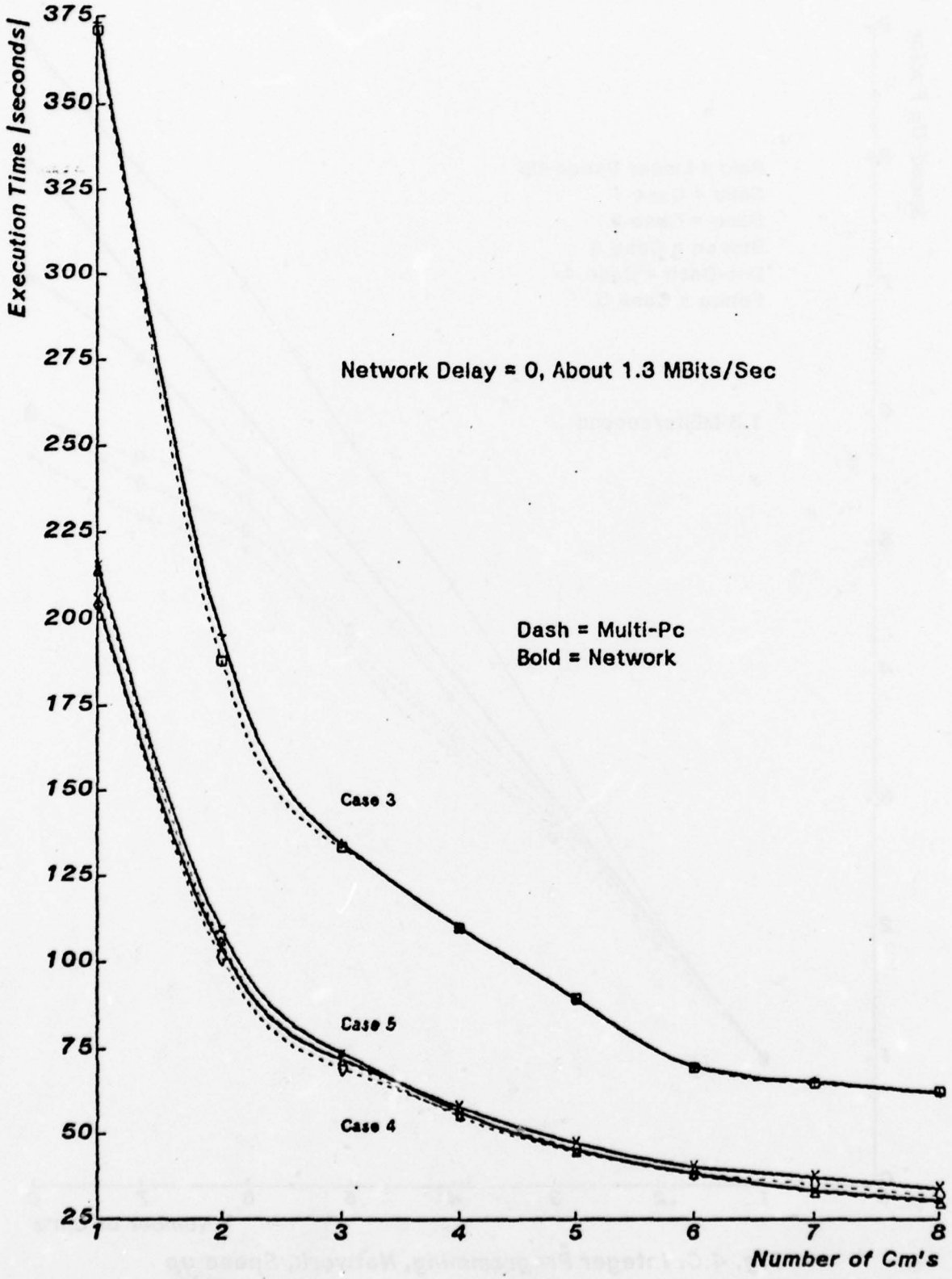


Fig. 4.5: Integer Programming, Comparing Network and Multi-Pc (b)

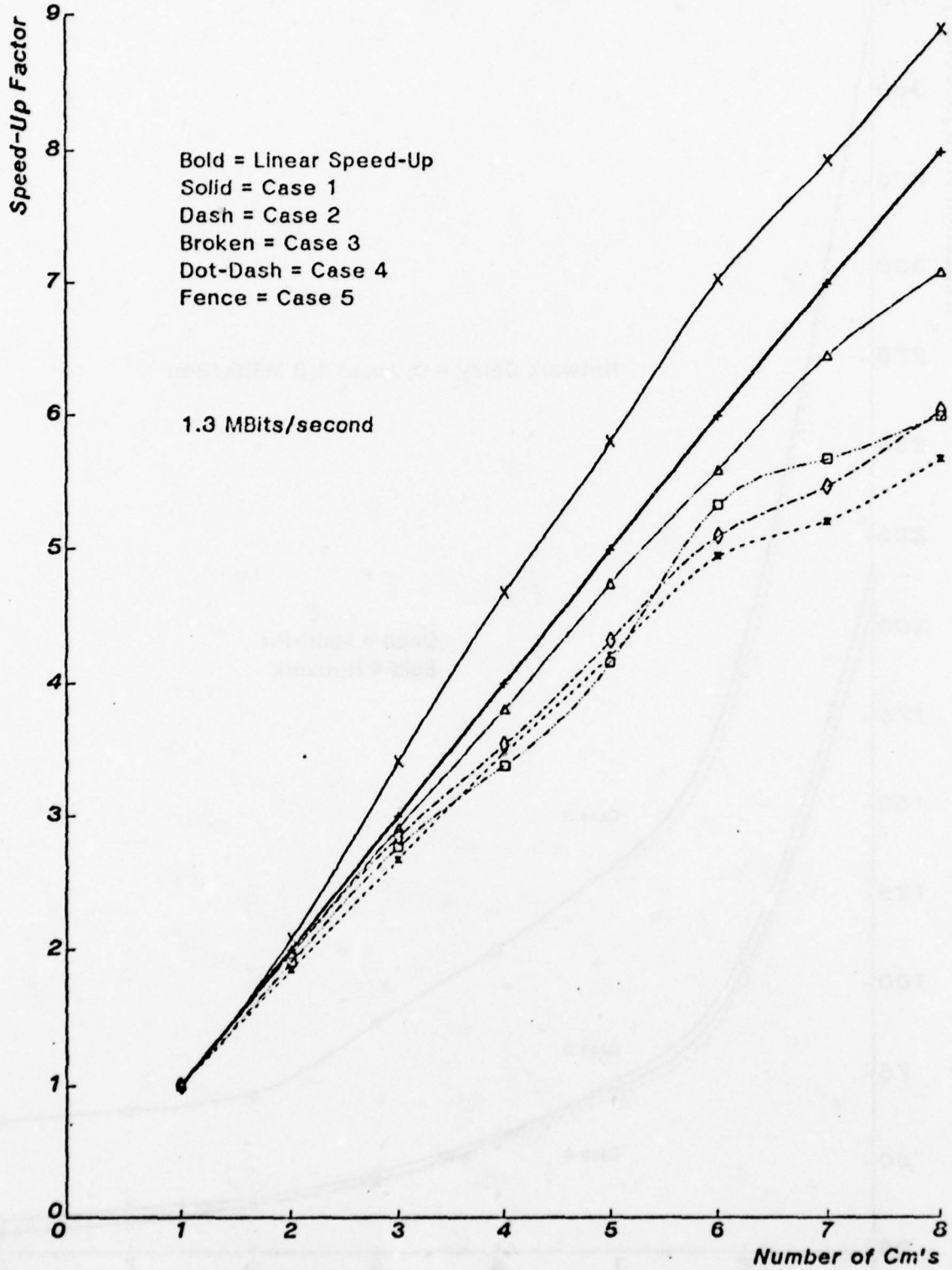


Fig. 4.6: Integer Programming, Network, Speed up

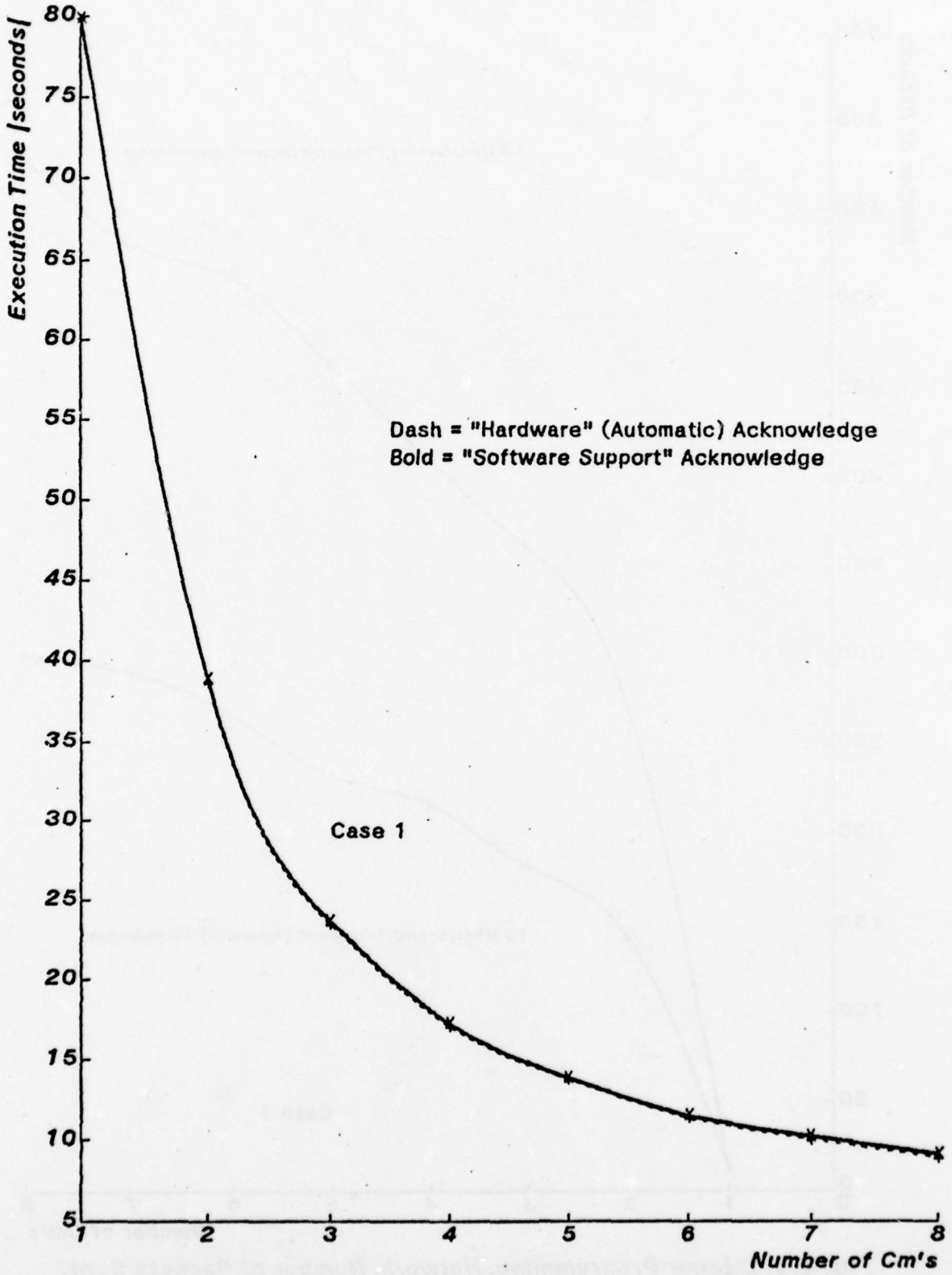


Fig. 4.7: Integer Programming, Effect of Acknowledge Mechanism

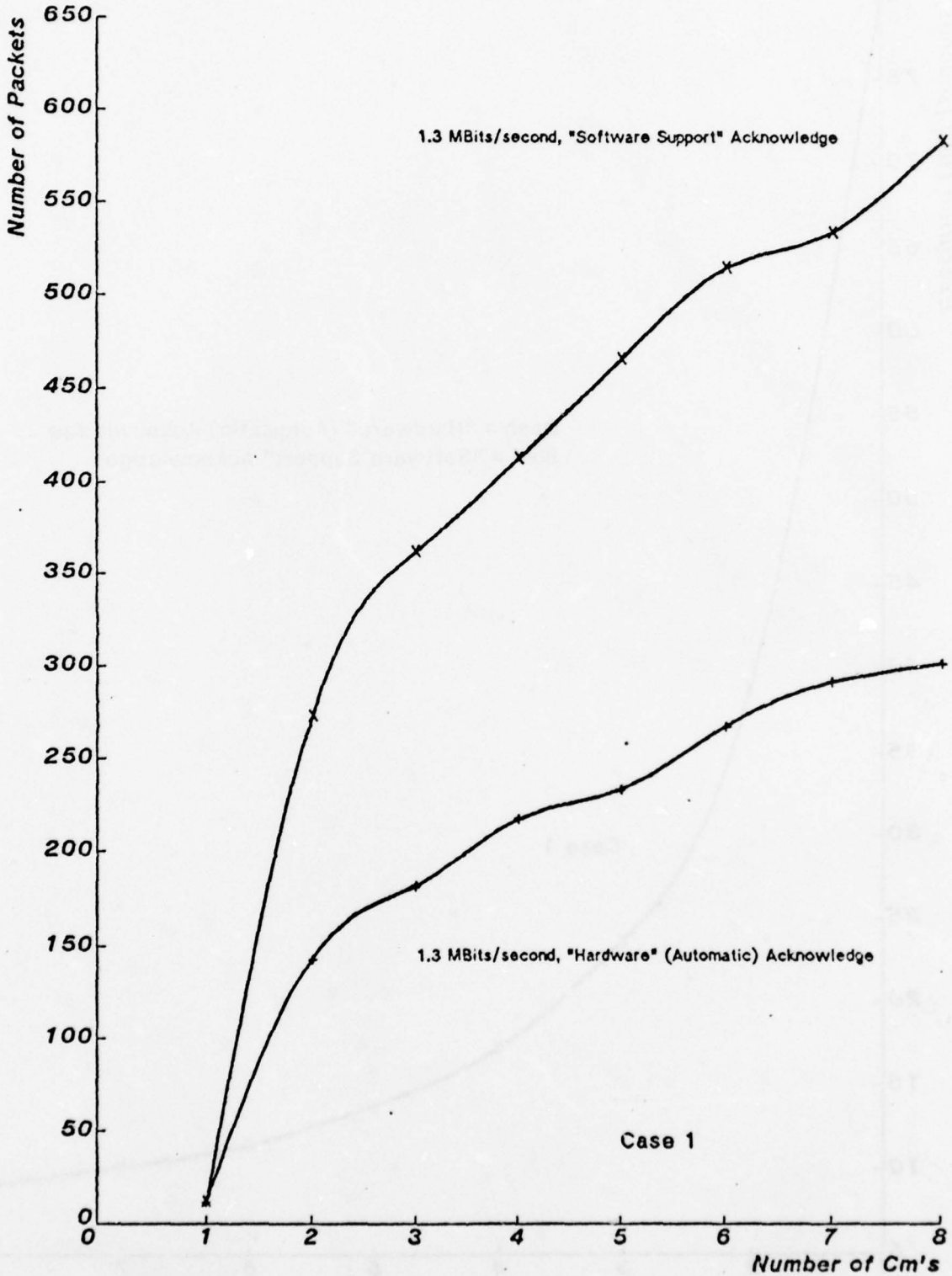


Fig. 4.8: Integer Programming, Network, Number of Packets Sent.

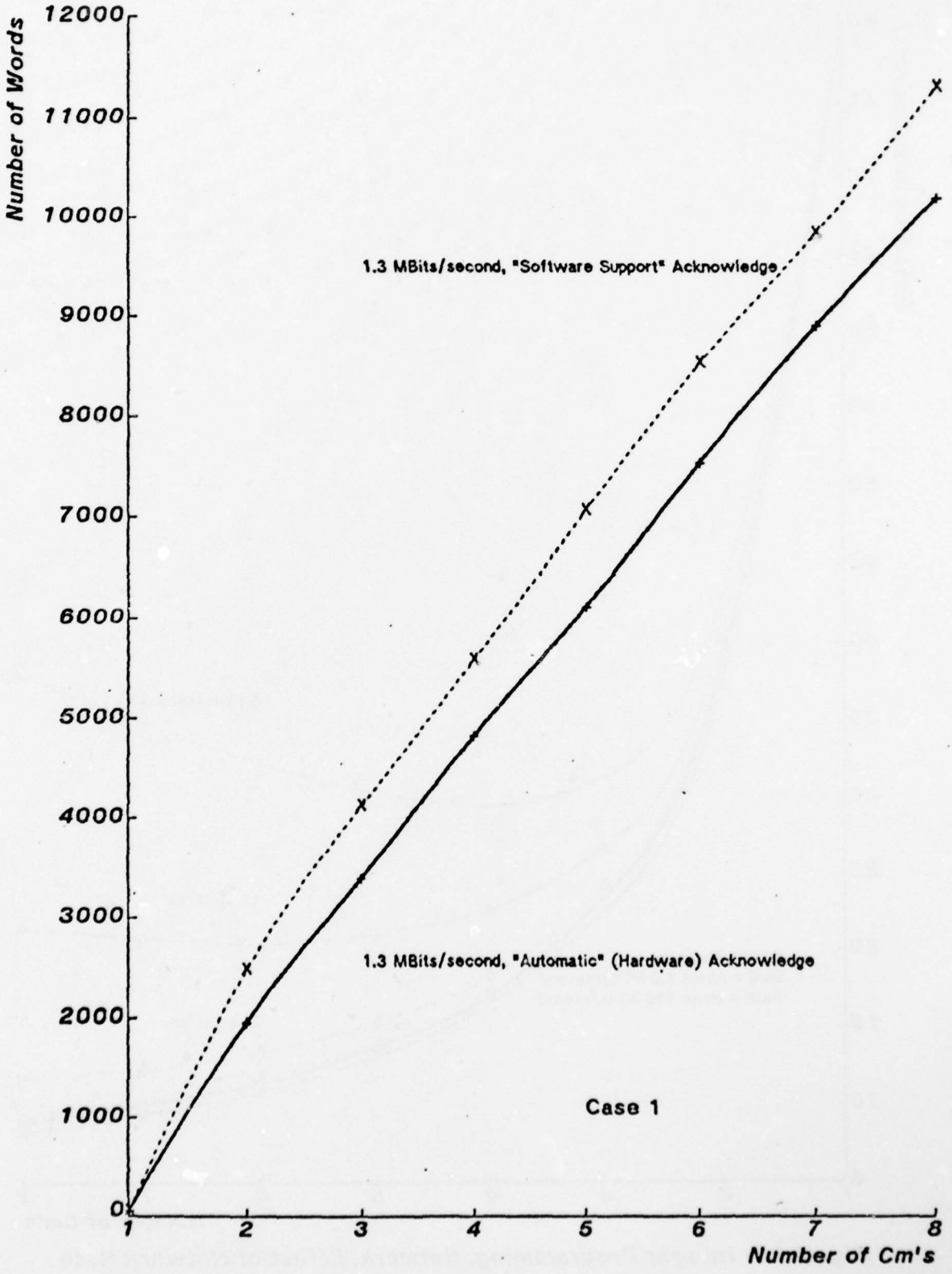


Fig.4.9: Integer Programming, Network, Number of Words Transferred

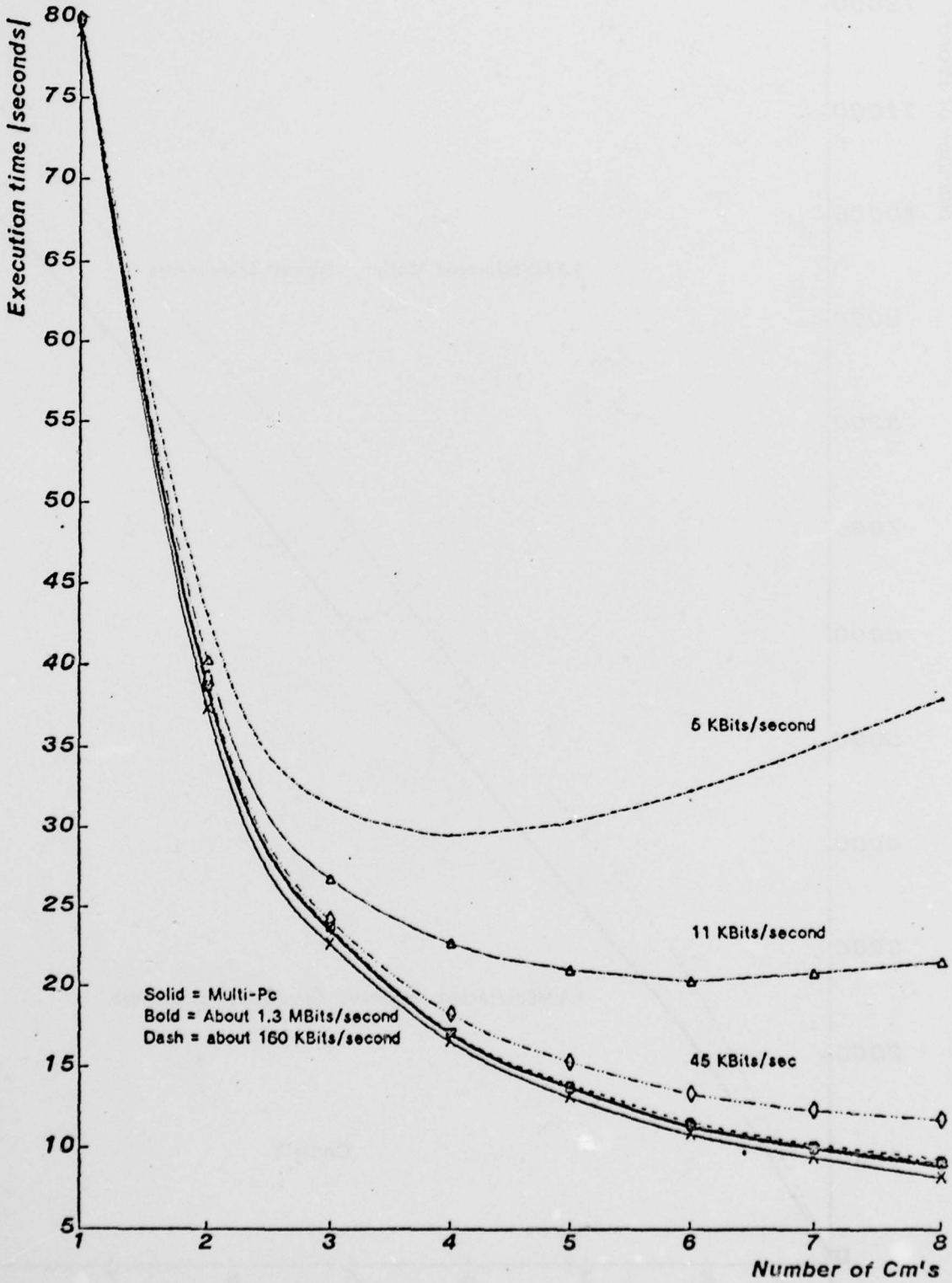


Fig. 4.10: Integer Programming, Network, Effect of Network Rate

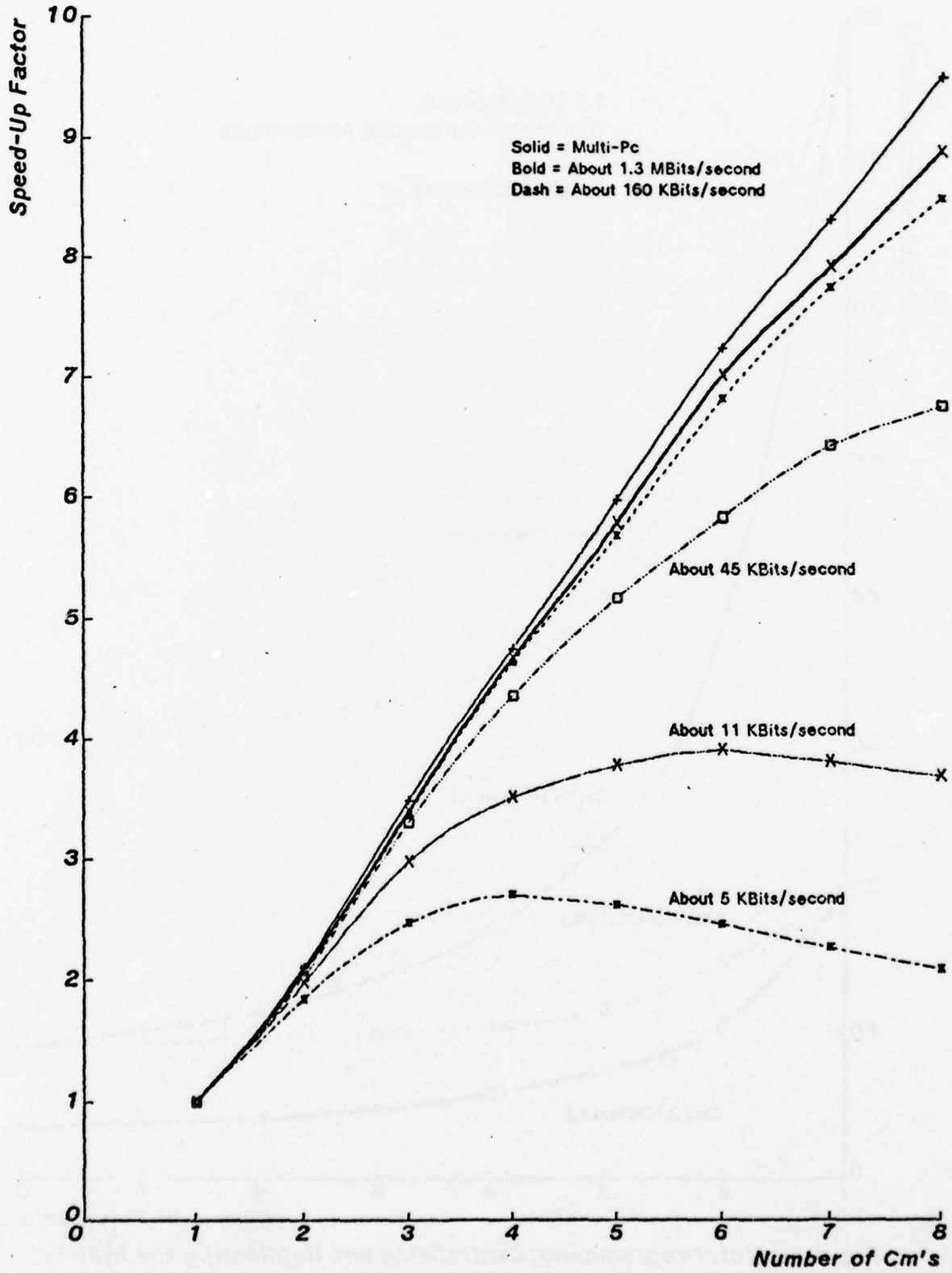


Fig. 4.11: Integer Programming, Effect of Network Rate on Speed-Up

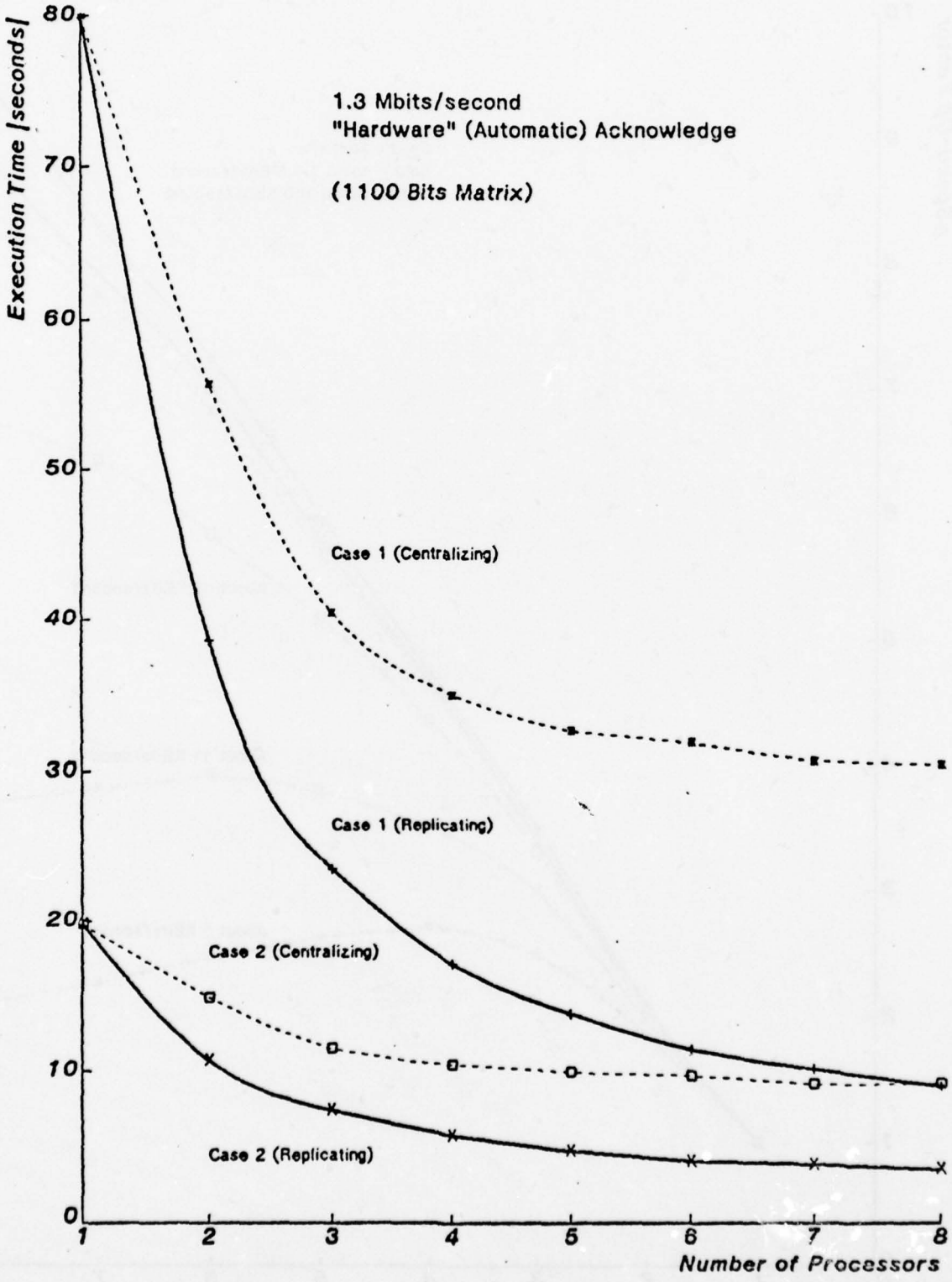


Fig.4.12: Int. Programming, Centralizing and Replicating the Matrix

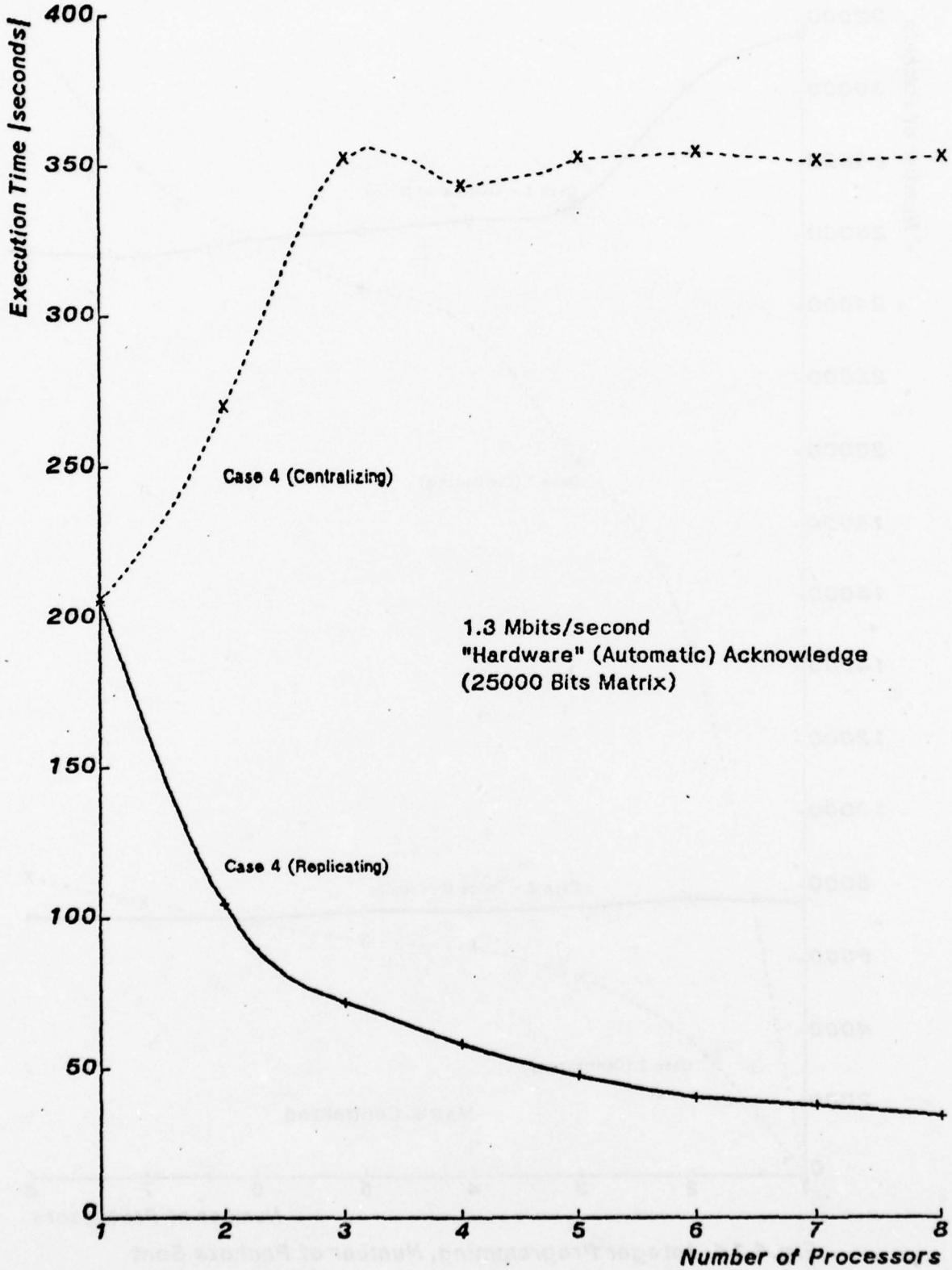


Fig.4.13: Int. Programming, Centralizing and Replicating the Matrix

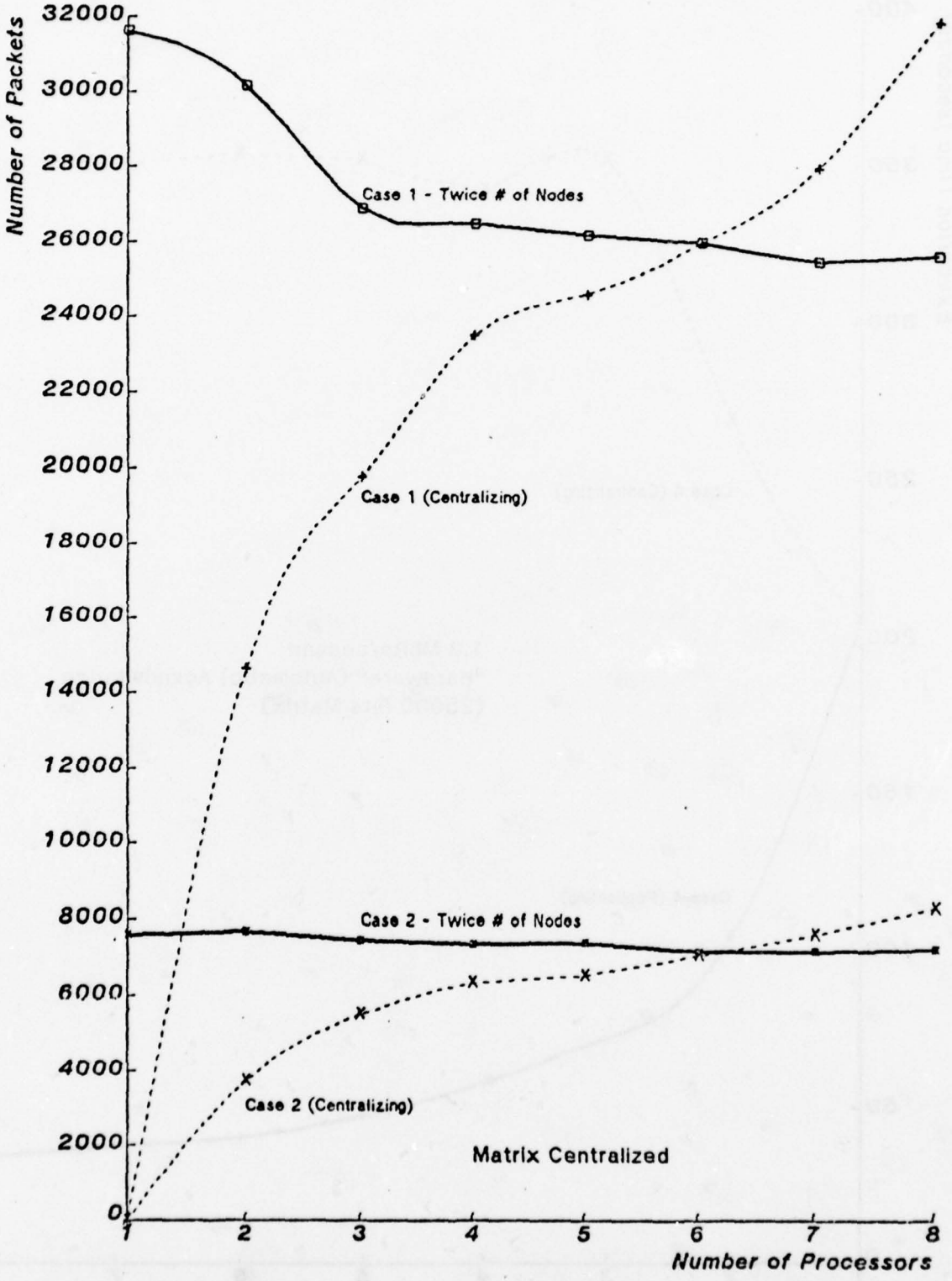


Fig.4.14: Integer Programming, Number of Packets Sent

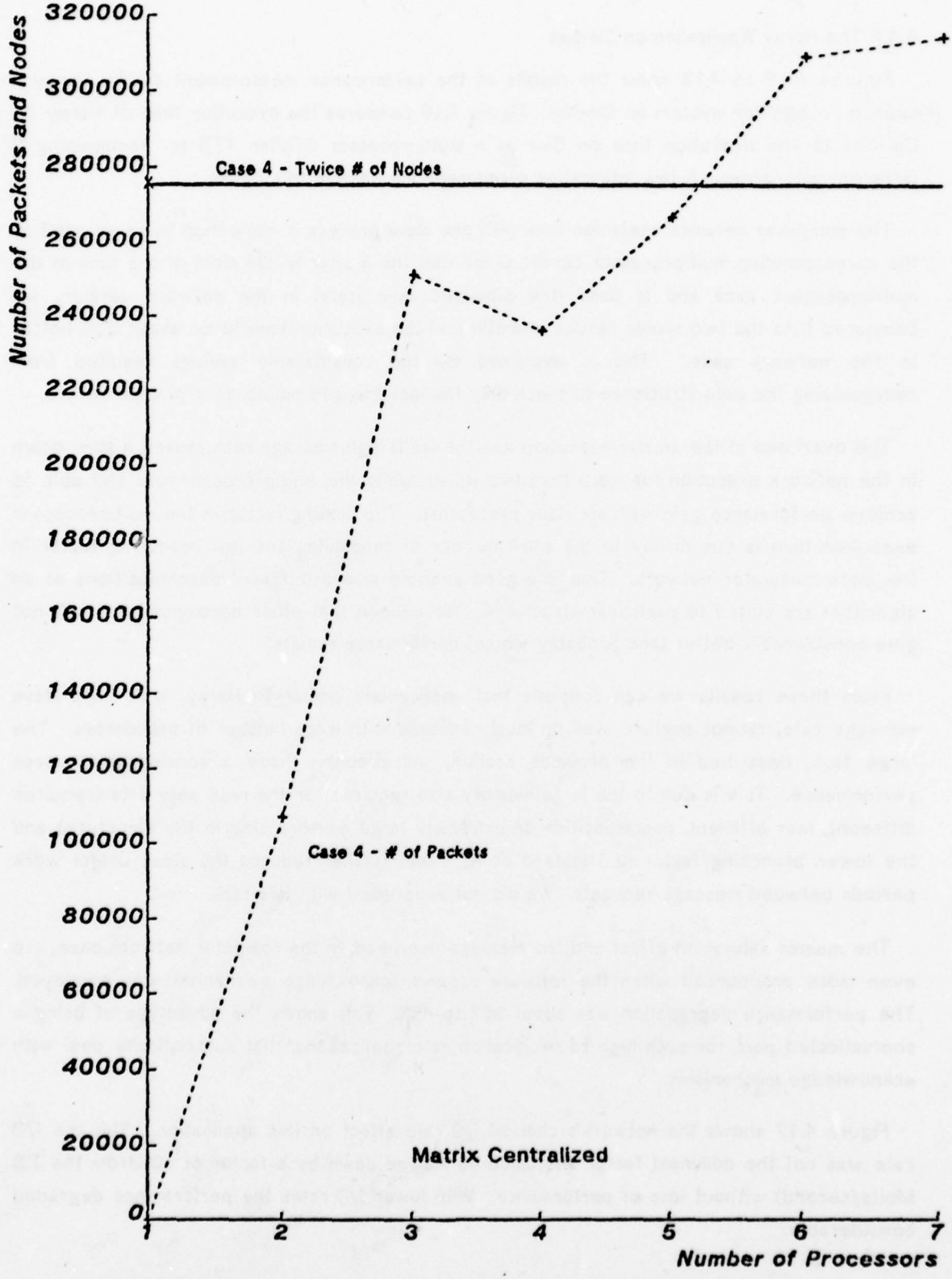


Fig. 4.15: Integer Programming, Number of Packets Sent

4.4.2 The Harpy Application on Cm-Net

Figures 4.16 to 4.18 show the results of the performance measurement of the Harpy - speech recognition system on Cm-Net. Figure 4.16 compares the execution time of Harpy on Cm-Net to the execution time on Cm* as a multiprocessor ([Feiler 77]) for recognizing 3 different utterances. A few interesting phenomena are seen in this figure.

The computer network execution time with one *slave* process is more than twice as good as the corresponding multiprocessor curve!. Given that the *master* is idle most of the time in the multiprocessor case and is used (the pipelined *save state*) in the network version, we compared it to the two slaves results and still find the execution time to be about 30% better in the network case. This is explained by the considerable savings resulted from reorganizing the data structures to check only the last changed nodes, as explained before.

The overhead in the *master* execution and the (still) high message rate caused a slow down in the network execution for more than two *slaves*, while the multiprocessor was still able to achieve performance gain with six *slave* processors. The limiting factor in the multiprocessor execution time is due mainly to the small number of candidates and low branching factor in the desk calculator network. This is a good example where different decompositions of an algorithm are suited to particular structures. We believe that other decompositions will not give considerably better (and probably worse) performance results.

From these results we can conclude that applications similar to Harpy, with high *slave* message rate, cannot perform well on local networks with large number of processors. The large task, described in the previous section, will probably have a considerable worse performance. This is due to the large memory size required for the *read only* data (requires different, less efficient, decomposition or extremely large memory size in the structure) and the lower branching factor (2.9 instead of 4) - that further reduces the *slave* useful work periods between message requests. We did not experiment with this task.

The *master* saturation effect and the message overhead, in the computer network case, are even more pronounced when the *software support* acknowledge mechanism was employed. The performance degradation was about 35% to 45%. This shows the advantage of using a sophisticated *port*, for such high communication rate applications, that automatically deal with acknowledge mechanisms.

Figure 4.17 shows the network's channel I/O rate effect on this application. Still the I/O rate was not the dominant factor and could be slowed down by a factor of 10 (from the 1.3 Mbits/second) without loss of performance. With lower I/O rates the performance degraded considerably.

Figure 4.18 presents the number of packets that were sent during the Harpy execution. The rate was about hundred times higher than in Case 1 of the Integer Programming application. Most of the packets are being sent when one *slave* is used (for getting the next assignment, transfer of search results etc.). The linear grows, with increased number of processors, is explained by the synchronization messages that are being sent to all the participating processes in the end of each segment of the utterance.

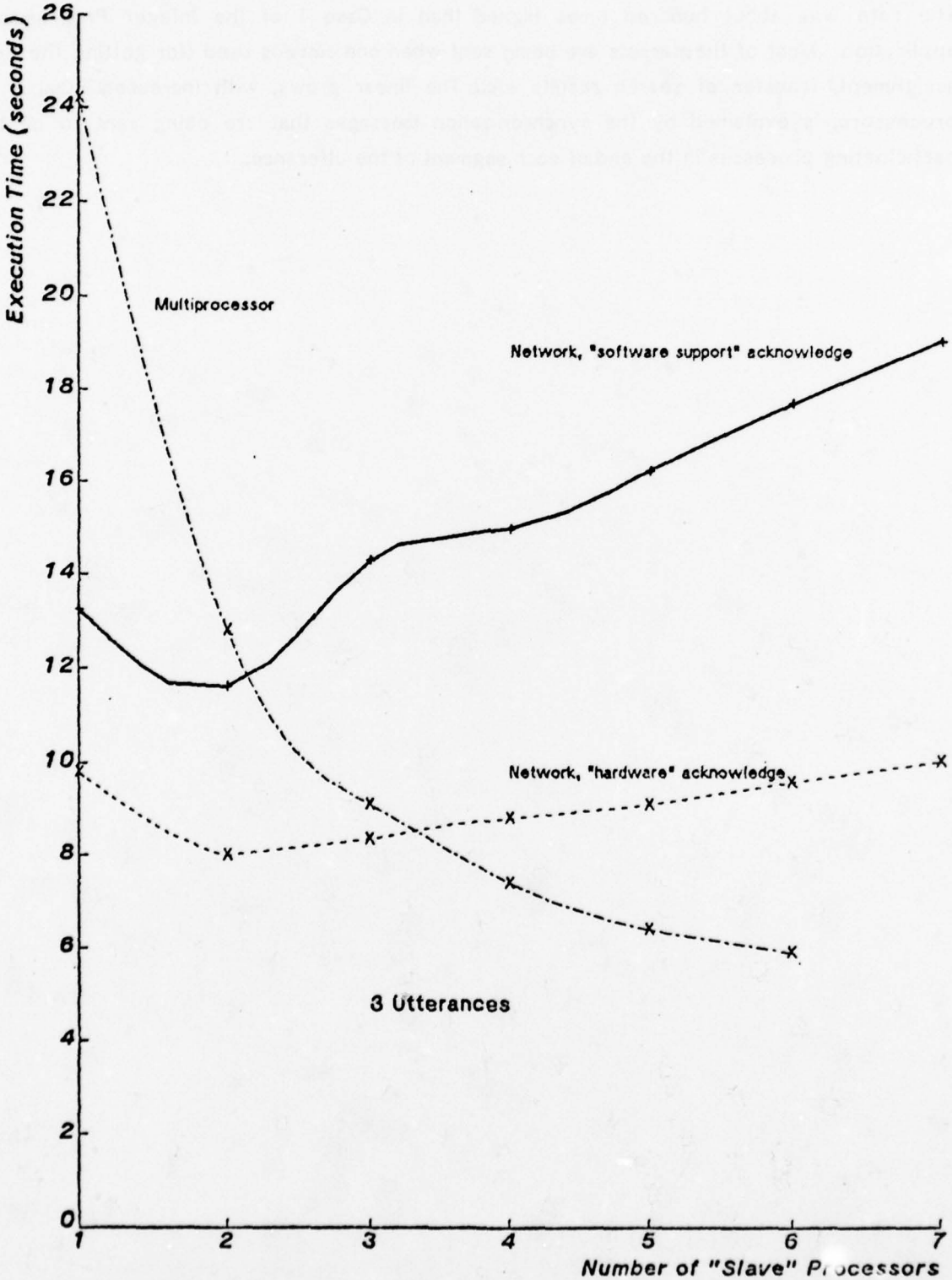


Fig. 4.16: Harpy, Execution Time of Multiprocessor and Network

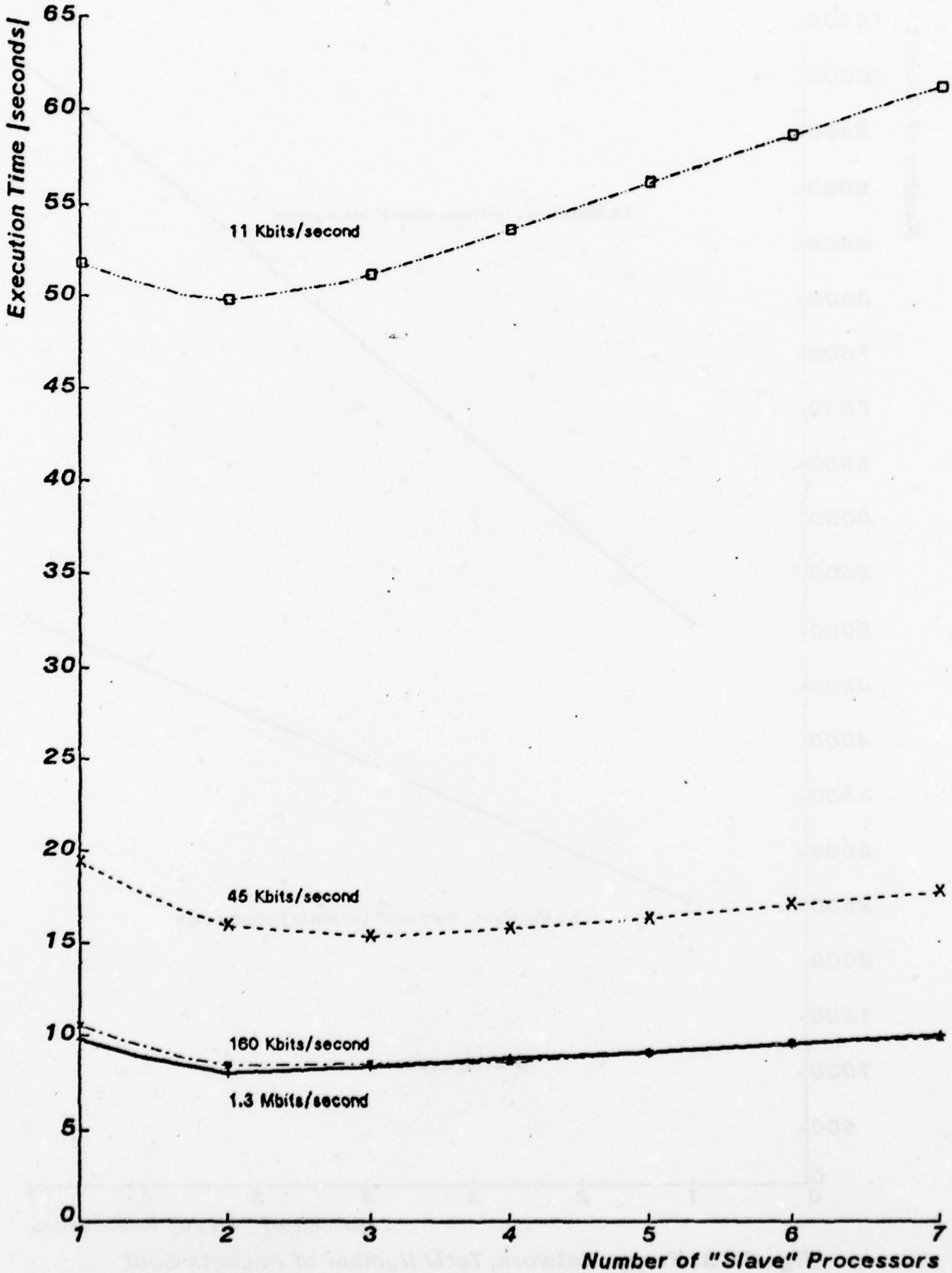


Fig. 4.17: Harpy, Network, Effect of Network Rate on Execution Time

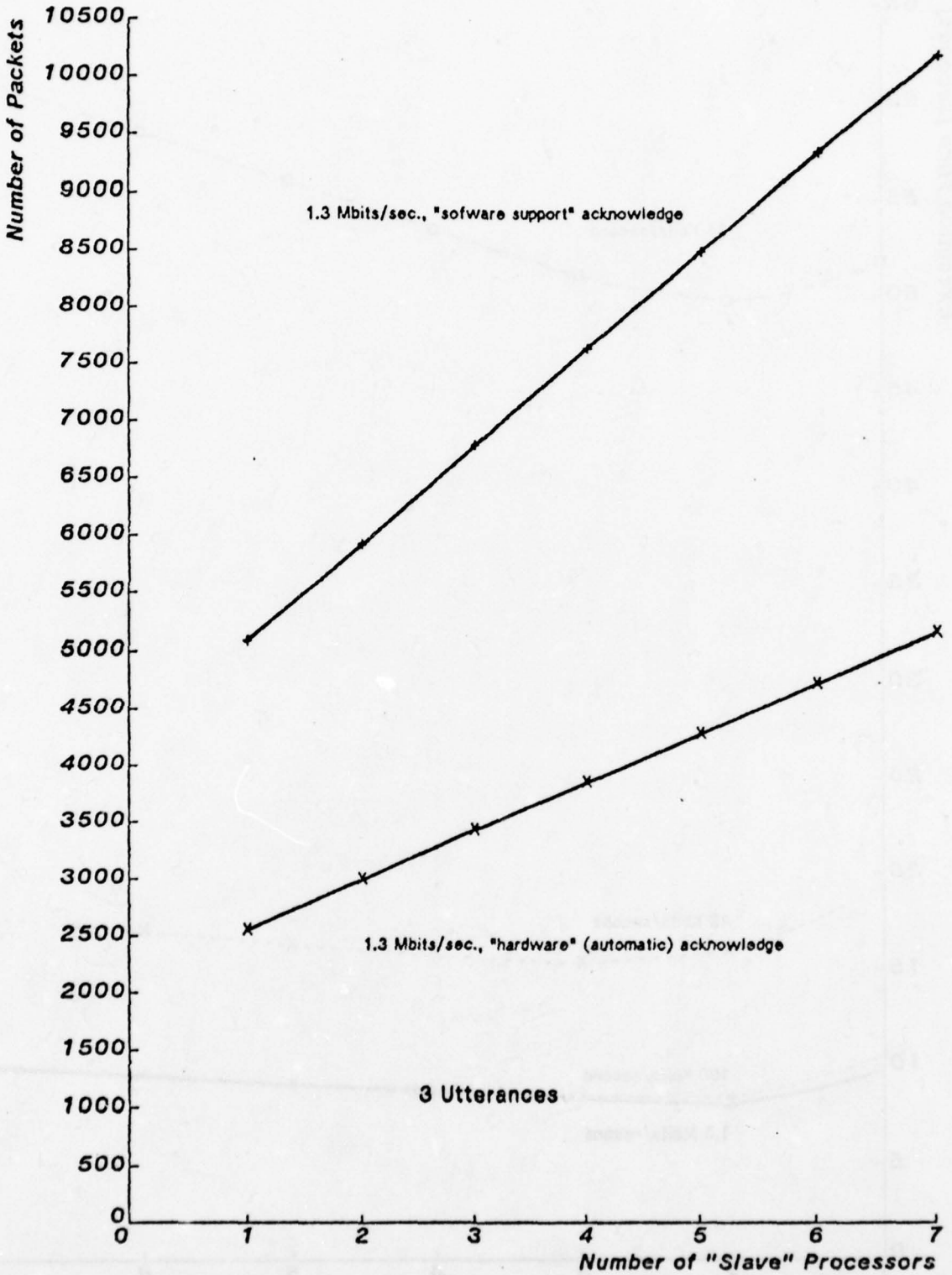


Fig. 4.18: Harpy, Network, Total Number of Packets Sent

4.5 Performance Model of a Local Computer Network

4.5.1 Introduction

As mentioned in section 4.1 the performance models of computer networks, that were used in the past, dealt with the design and evaluation of the network transport, its topology, protocol and capacity. None of these models was able to approximate the performance behavior of the applications (that were described in the last sections) and thus they were not useful for our purpose. In this section we develop a simple performance model that shows the performance of an application which utilizes the local computer network to acheive performance gain.

A recent paper by Kinney and Arnold [Kinney 78] was the only model somewhat similar to our need. In their model they have assumed independent subtasks executing on a multiple processor structure with a shared global bus. Each subtask requiring a cyclic processor execution and bus usage. Their objective was to determine the processing power of such a system, and used linear assumptions and a simple queuing model to obtain upper limits on the processing power and bus usage.

This model was not sufficient to represent the scheduling and message transfer requirements of an application. Observing the measurement results we have derived nine parameters that we believe dominate the local network performance. The nine parameters are presented below. The first three are the parameters that mainly characterize the local network, while the last six mainly characterize the application. These parameters are:

1. Transport mechanism communication rate and port message overhead.
2. Acknowledge mechanism used.
3. Number of processors in the local network.
4. The *slave* mean execution time between successive requests (we assumed here application programs that are characterized by a cyclic execution time followed by a message to the *master* requesting required information).
5. The *master* mean response time to *slave* requests.
6. Message overhead in *master* and *slave*.
7. Wether the *master* participate in the task as a slave or not.
8. Number of initialization packets and words for each addition of a *slave* process.
9. Average number of words in the *slave* and *master* messages.

Approximation to these parameters might be derived either from simple measurements (using one *slave* processor) or from estimation of the application behavior and code involved in the different tasks.

This list of parameters is not sufficient to describe the application behavior thoroughly, and deliver only a gross estimation of the behavior of that class of applications. As commented in chapter 1, modelling the exact behavior of of an algorithm is a difficult task to do, and is not general enough to generalize from. For example, we assumed the performance of the application to be directly proportional to the amount of processing available in the structure to process the application, while in many cases the application does not possess that property (e.g. does not have enough inherent parallelism to fully utilize the structure). We assumed here that an approximation of the application program behavior and of the local network parameters, as given above, should be sufficient to give a first order approximation of the network performance and usage for accomplishing the task.

4.5.2 The Performance Model

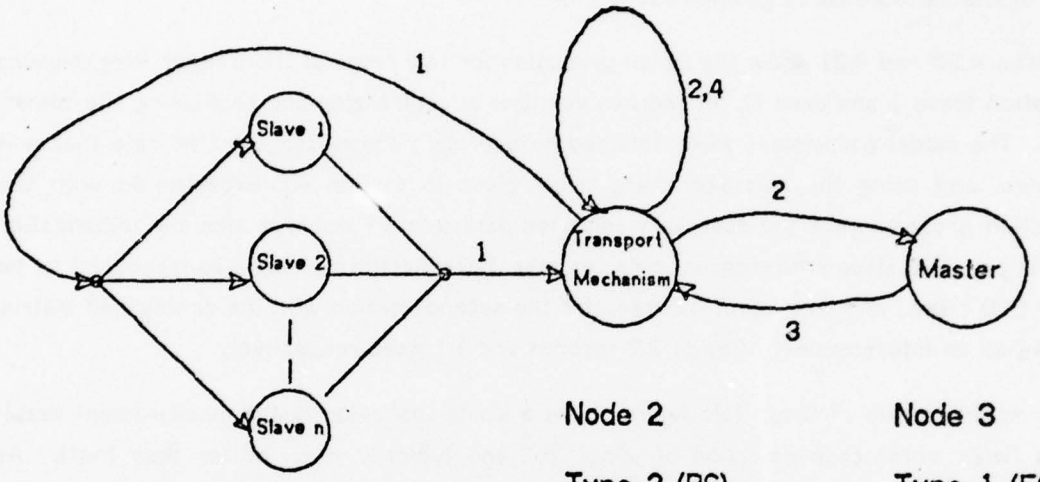
A simple, three node, queueing network model with four classes of customers was developed to model the local computer network behavior.The model is shown in figure 4.19.

Following an execution time in the *slave* a message request is transferred to the network transport channel. After some delay (and acknowledge delay if applicable) it was queued to receive service at the *master* node. The service time of this node includes the message overhead time. A return message is sent back from the master, through the transport mechanism (again with some delay), to the waiting processor.

The service times of the *master*, *slave* and I/O transport were derived based on the nine parameters given above; That is, the network transport mean service time for message requests of certain types was determined by the I/O rate, average message size, message overhead, and acknowledge mechanism costs. The initialization phase was modelled by incorporating a linear time model which calculates the initialization time in the beginning of the execution.

Using the efficient computation algorithms described in appendix 3 (the same algorithms that were used for the multiprocessor modeling) an APL program was written to calculate the utilization of nodes 1 and 3. These utilization results where the basis for the network performance predictions given below.

4.5.3 Comparing Model and Measurement Results



With "Software" Acknowledge	No Acknowledge
$P_{1,1;2,1} = 1$	$P_{1,1;2,1} = 1$
$P_{2,1;2,2} = 1$	$P_{2,1;3,2} = 1$
$P_{2,2;3,2} = 1$	$P_{3,2;2,3} = 1$
$P_{3,2;2,3} = 1$	$P_{2,3;1,1} = 1$
$P_{2,3;2,4} = 1$	
$P_{2,4;1,1} = 1$	

Figure 4.19: Local Network Queueing Model

modnet.spx/d

Trying to validate the accuracy of the model, we made a series of experiments to measure the nine network parameters for the two applications, and to compare the predictions of the model to the measurements given in section 4.4.

Figures 4.20 and 4.21 show the model prediction for two cases of the Integer Programming application (case 1 and case 4), for the two versions of centralizing and replicating the binary matrix. The model parameters were obtained by averaging the message traffic rate that was measured, and using the message timing delay given in section 4.2. Experiments with the application program gave the necessary remained parameters - message size and initialization information. The *slave* inter-request time for case 1 (replicated matrix), was measured to be slow - 500 msec., and was about 4.5 msec for the second version with the centralized matrix. Case 4 had an inter-request times of 2.2 seconds and 1.1 msec respectively.

The model results of those two figures have a similar behavior to the measurement result curves (with worst case deviation of about 25% and typically much better than that). As mentioned before the model cannot predict anomalies and deficiencies in the algorithms and assumed that the execution time degrades only as a factor of the processors idle time during messages transfer. The model showed that centralizing the matrix caused the master process to be the bottleneck in the system with the measurement overhead time being responsible for considerable utilization loss.

Figure 4.22 show the I/O rate influence on performance, as predicted by the model (dashed curve) compared to the measurement result (bold curve). Here again we see that the model has predicted a similar behavior to the measurements. The surprising pessimistic prediction of the model mainly results from the better than linear speed-up of case 1 of this application. The model showed that the slow down with low I/O network rate and many processors is mainly due to the initialization phase in the beginning of the execution.

Figure 4.23 shows the results for the Harpy application with several I/O rates. It is encouraging to see that the model was able to predict the performance behavior of such a complex algorithm with reasonable accuracy. The model give more optimistic results than measured (as expected) and the difference increases when more processes are participating in the task - as the Harpy (Descal) algorithm, with its small number of candidates per segment of speech and small branching factor, cannot support many processes even in the multiprocessor version. Harpy's *slave* inter-request time was 6 msec., and the *master* mean service time was averaged to 4.5 msec.

In conclusion these measurement results show that the performance model can predict, with reasonable accuracy, the gross behavior of an application on top of local computer network structure.

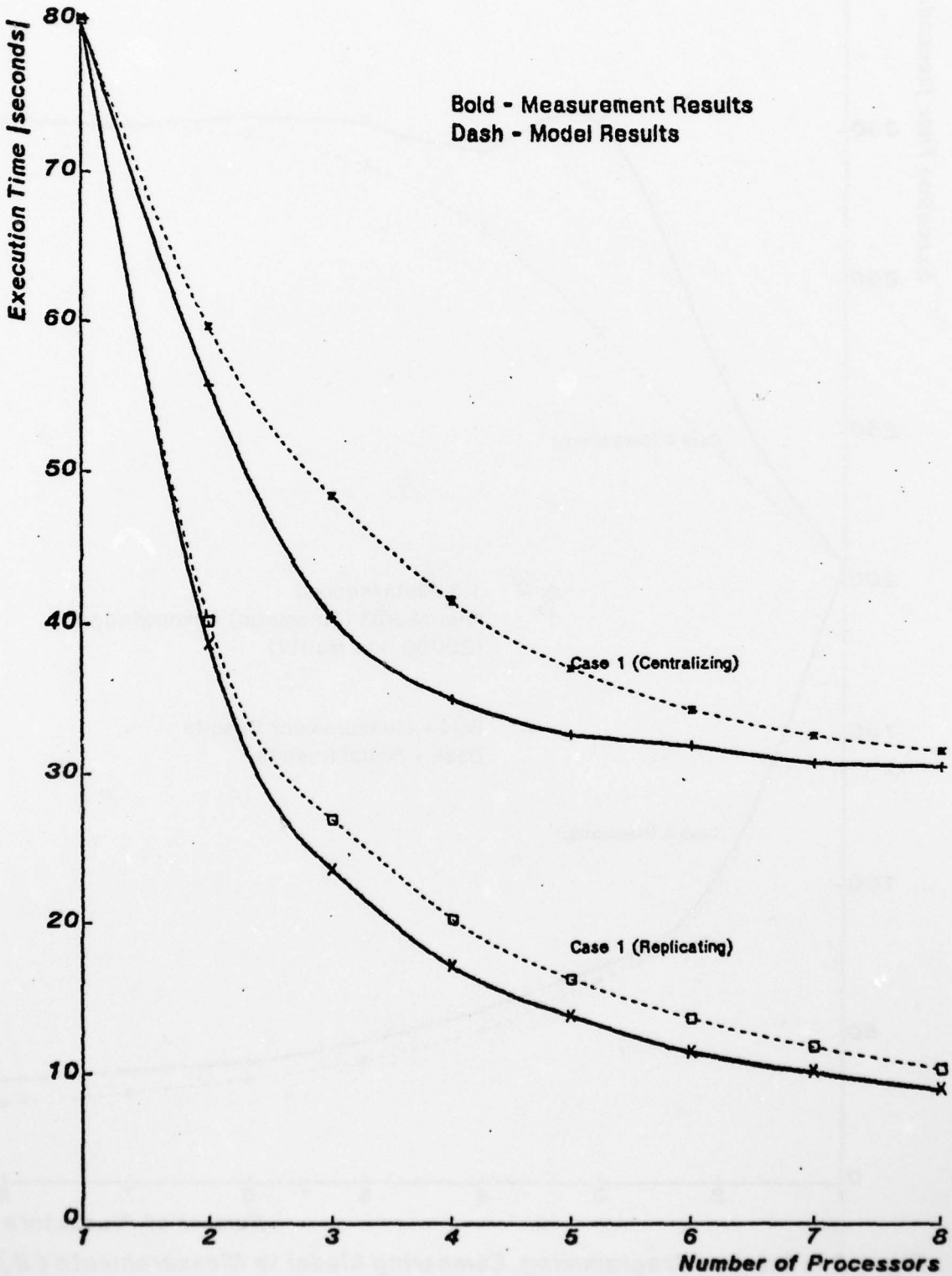


Fig.4.20: Integer Programming, Comparing Model to Measurements (1)

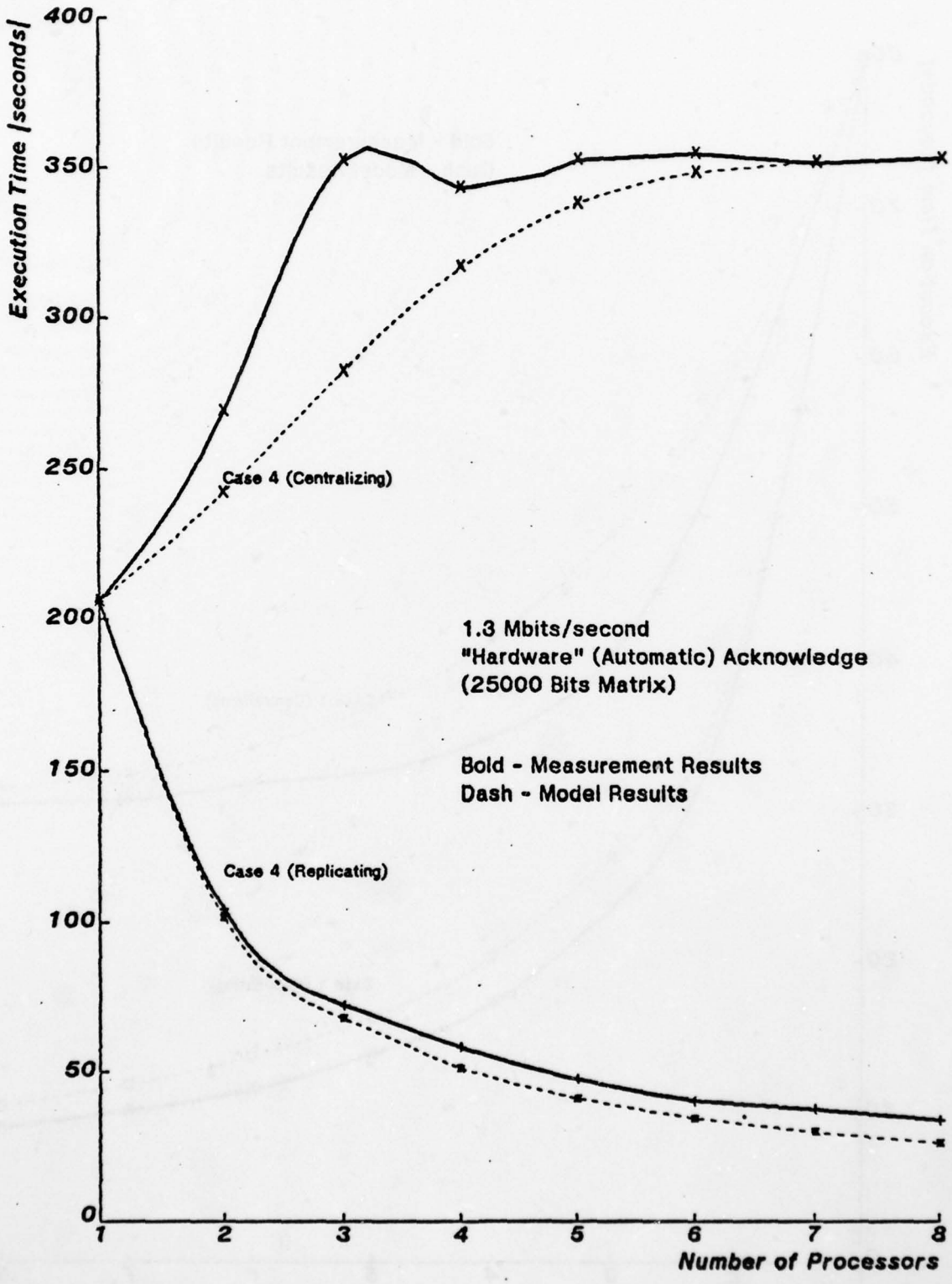


Fig.4.21: Integer Programming, Comparing Model to Measurements (2)

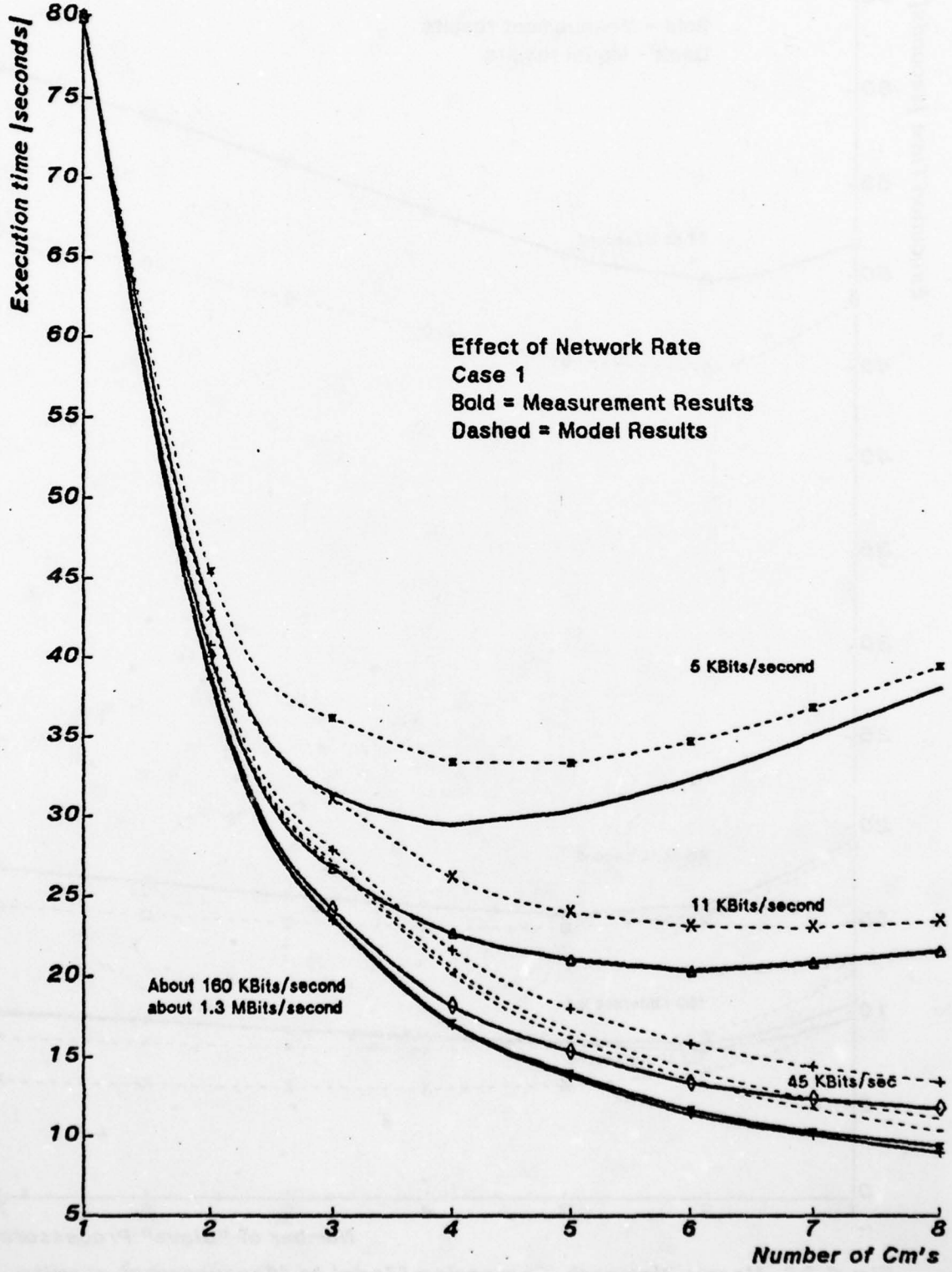


Fig.4.22: Integer Programming, Comparing Model to Measuremonts (3)

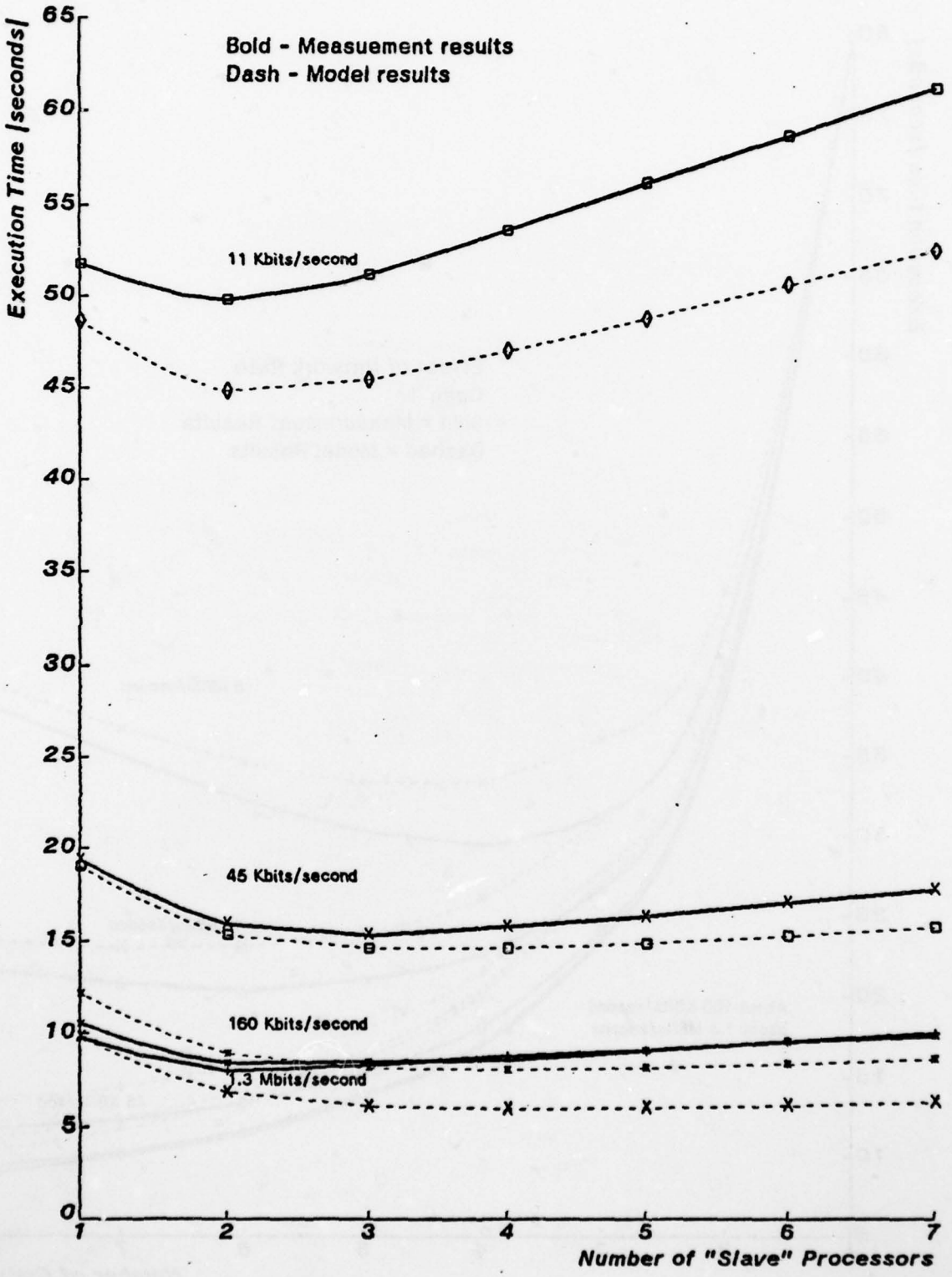


Fig.4.23: Harpy, Network, Comparing Model to Measurement results

4.5.4 Model Predictions

Gaining confidence in the model capabilities, we applied the model to investigate several performance questions that we did not, or were not able, to measure. Figure 4.24 show the model prediction of the local computer performance with up to 50 processors and several communication rates. The parameters were those of Case 1 of the Integer Programming (with replicated matrix). The graph show the saturation effect of the transport mechanism when more processors are added. For this case the network was still capable of giving a theoretical speed up of about 22 for 35 processors. Applying the same criteria as was applied in the multiprocessor investigation (in chapter 3) of the 50% performance return factor (i.e. adding a processor only if it returns 50% of its maximum capability) estimated this point to be about 28 processors in the 50 Mbits/second case, 22 processors for 1.3 Mbits/second, and 15 for 340 Kbits/second. The slow down is mainly due to the increasing percentage of the initialization time (i.e. replicating the *read only* data) in the beginning of the execution when more processors are added.

Figure 4.25 show the model prediction of the acknowledge mechanism. The acknowledge message overhead in the case of many short messages (as is the case in the centralized matrix case) show that an automatic acknowledge mechanism is a must in that case. The acknowledge has a negligible effect in low message rates (some of this predicted points were checked for validation and found to be better than the worst case mentioned before).

Figure 4.26 shows a comparison of the models results for the three versions of the Integer Programming application - a one cluster, multiprocessor version (with matrix centralized) and two local computer network versions (with matrix replicated and centralized, as discussed before). The models predicted the performance of the two structures with up to 50 processors in each. The figure gives a clear presentation of the advantage achieved by sharing the address space (in the multiprocessor case) with minimal overhead. Although the Integer Programming application has an extremely low message communication rate (in the matrix replication case) the 50 Cm's, multiprocessor version had more than twice higher performance than the best local computer network version, while requiring less memory space. Matrix centralization, in the local computer network case, require the same amount of memory, but has a high message communication rate that saturates the *master* process almost immediately.

Figure 4.27 shows another example of using the performance model for performance prediction. Another aspect - the effect of different classes of applications with differing rate of message transfers, is shown in this figure. The parameters used for the model where the same as those for the integer Programming application while the *slave* inter request message

time and the transport mechanism communication rates were varied. A network of 10 *slave* processors and one *master* processor was assumed. The curves show a critical rate of message transfer above which the performance degrades rapidly. This figure, together with figure 4.24, form a three dimensional space with the performance in one axis, the message rate and number of processors in the other two axes, and with surfaces of varying network transport communication rates. Desired criteria for efficient operations of classes of applications on a given structure, or the suitability of a proposed structure to solve an application efficiently can be derived from such investigations.

The peculiar behavior of the curves in the high message rate region (region of performance improvement with the lowering of the network transport communication rates) is explained by the participation of the *master* in the task. Lower transport rate (lower message rate from *slaves*) allows better utilization of the *master* to participate in the task. Long initialization time in the beginning of the task, in the other hand, causes a lower speed-up factor for very slow communication rates. The two coinciding curves - 1.3 Mbits/second and 10 Mbits/second show that the transport mechanism communication rate, in that case, is not a limiting factor for rates above (about) 1 Mbits/second.

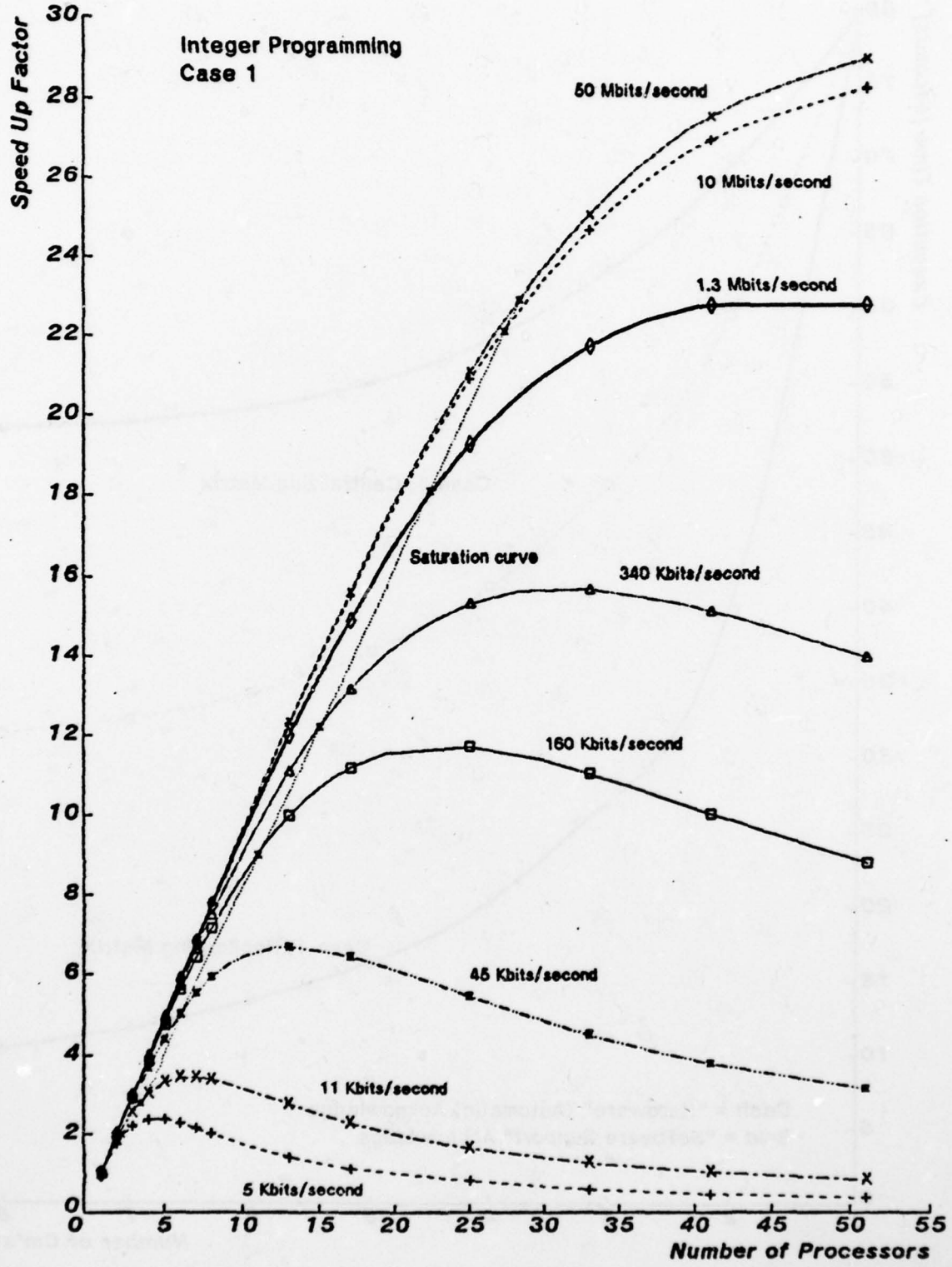


Fig.4.24: Integer Programming, Model Prediction of I/O Rate Effect

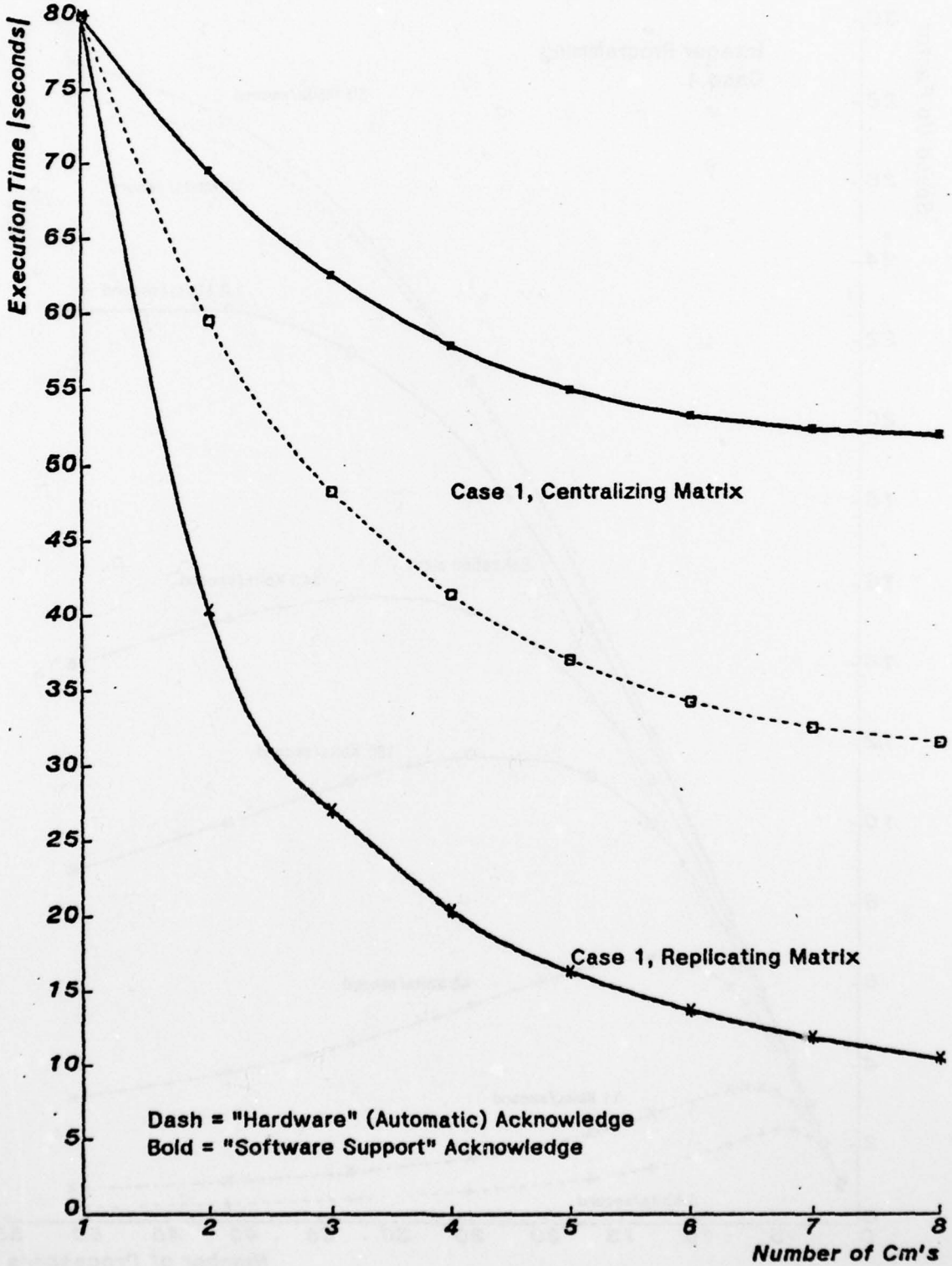


Fig.4.25: Int. Programming, Model Prediction of Acknowledge Effect

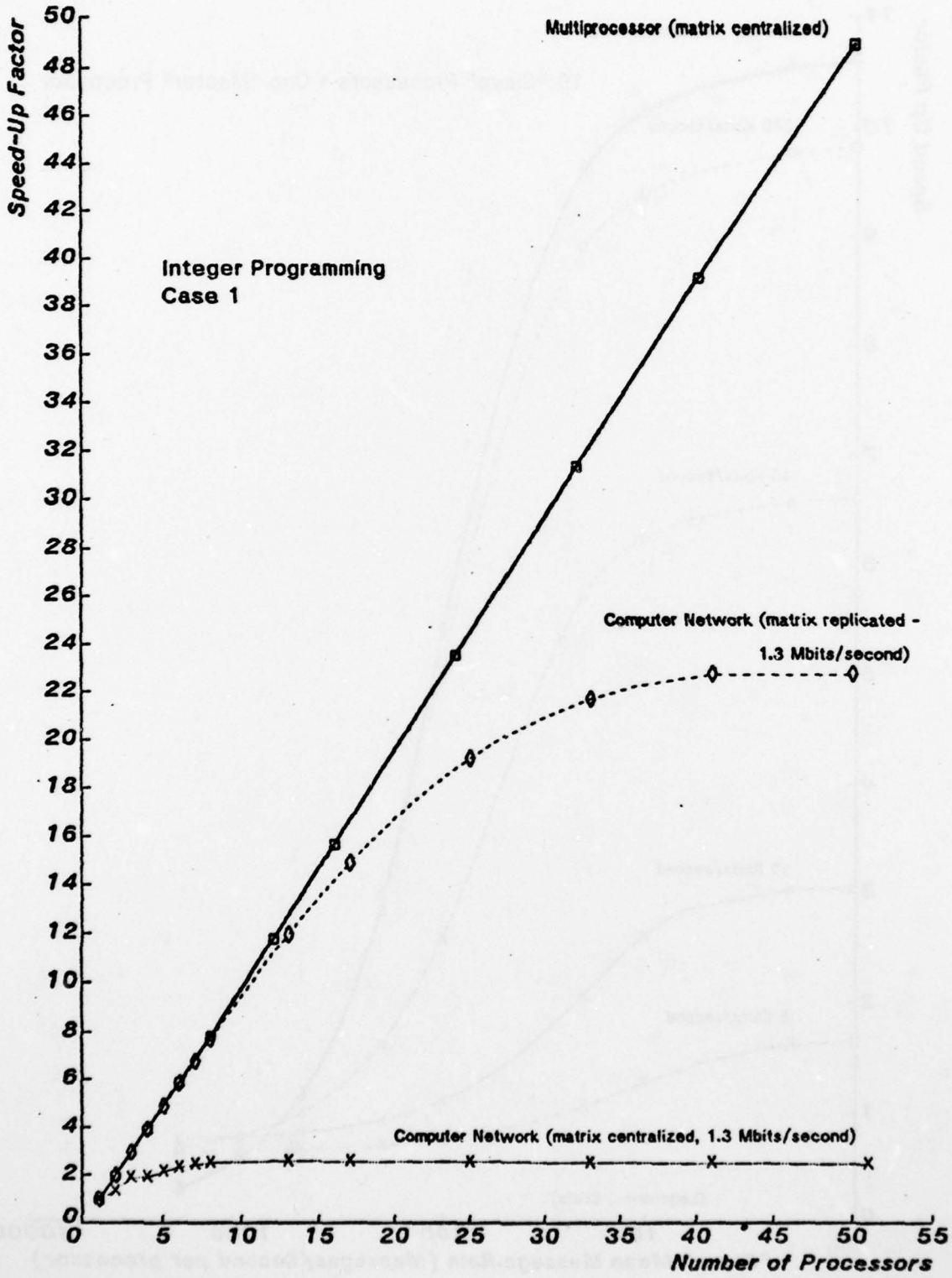


Fig.4.26: Model Prediction, Comparing Multi-Pc to Local Networks

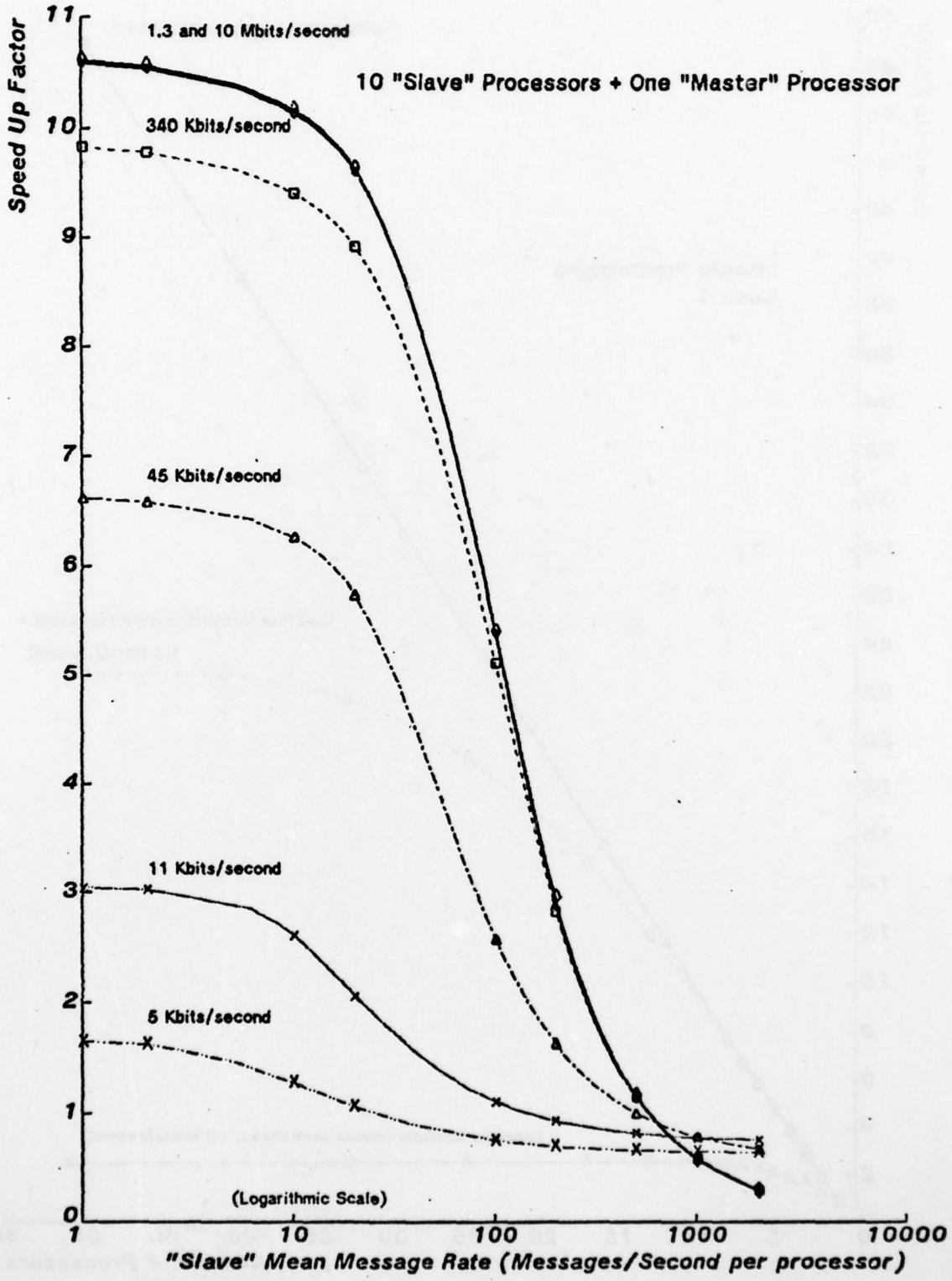


Fig. 4.27: Model Prediction of Classes of Applications

4.6 Summary and Conclusions

In this chapter the performance of multiple computer-modules, connected as a local computer network, were discussed and the main attributes which affect the performance were investigated. In moving from constructing applications on multiprocessors, where all processes (and processors) share the same address space, to the use of local computer networks with messages as the inter-process communications means, the emphasis has moved from trying to maximize the *hit ratio* to the local memory into minimizing the message rate between processes.

We have microprogrammed Cm* - the multiple processor structure - to act as a local computer network, *process interface software support* package was built on top of this structure to enable experimenting with application programs. Two application programs: Integer Programming and Harpy - Speech Recognition system, were used to evaluate the performance issues. The decomposition of the Integer Programming was similar to the multiprocessor version (with the obvious changes in message communication) and its performance, with matrix replication, was excellent. It was encouraging to observe the existence of some classes of applications for which the message overhead is endurable.

The second version of the Integer Programming (centralizing the matrix in one module), as well as the Harpy application, were examples for which the sharing of the address space gave considerably better performance than the local computer network version. In the Harpy case we showed that different decompositions of the algorithm, better suited to the network structure, are needed to utilize the structure efficiently. The effect of the *transport mechanism* communication rate and the acknowledge mechanism were also explored.

Analyzing the the measurement results led to the development of a simple, three node, queueing network model with four classes of customers. Nine parameters were the input to the model, and by comparing the model to the measurement results, were shown to represent with adequate accuracy the performance of the application programs on top of the local computer network structure.

The model was used for the investigation of various performance issues which resulted in better understanding of the important parameters that affect performance. Several performance prediction investigations were done which illustrates that the performance is mainly a four parameter, three dimensional, problem; With performance, number of processors, and message rate between processors as the axes, and with surfaces of variable *transport mechanism* communication rates. The range of a suitable classes of applications for a given structure, or a suitable structure for an application, can be determined from such

investigations.

The decomposition versions of the Integer Programming illustrate the cost/performance (or alternatively memory size/execution time) trade-off - that is more apparent in the local computer network case, where the localization effect is even more pronounced than in the multiprocessor case.

The limited experience gained from experimenting with the application programs suggests that creating an application program, while emphasizing different requirements, was not vastly difficult than for a multiprocessor decomposition. The emphasize on low communication rate is a restricting factor, but the simplicity of the synchronization and *critical section* mechanisms in the local computer network case (only one processor can access a certain memory) alleviates the application implementation problem. Debugging the applications was found to be difficult, as a result of the autonomous processors and the inability to examine the memory of various processors directly. Good debugging tools are needed here.

This chapter deals with possible performance of applications on top of a local computer network structures and we regard it as a pioneer attempt to address this problem. We believe that the performance model is a first step in that direction. Chapter 5 reiterates the main results and elaborates on the main conclusions.

5. Conclusions and Future Research Areas

5.1 Summary and Main Contributions of the Thesis

In this dissertation the performance of multiple processor structures was studied. Multiple processor structures span a wide spectrum of structures with varying degree of coupling. Our interest lies in the performance of structures made out of computer-modules (i.e processor-memory pairs) connected either as a multiprocessor, with sharing of a single address space between all processors, or as local computer networks where explicit messages are the communication means between the processes executing on the processors in the structure.

Our results are based both on experiments and measurement of benchmark programs on actual multiprocessor and local computer network structures, and on performance models that were validated by the measurement results, and then applied for the prediction and analysis of various performance issues. Due to the little knowledge and experience in building and using multiple processor structures - we hope that this practical methodology and the methods, tools, and measurements conducted will advance the state of the art of this subject.

A description of the methodology used, as well as a short description of Cm*, a multiple micro computer-module structure, is given in chapter 1. Cm* is a flexible research vehicle that allows relatively easy emulation of various structures with varying parameters. We have used it to emulate both multiprocessors and local computer networks. The performance models, that were developed and used, were mainly at the PMS structure level. The performance investigation at this level is only one of the three main factors that affect performance of an application program on top of multiple processor structure; the other two are the overhead due to operating system interactions, and the parallelism in the algorithm of the application program itself. Both are subjects of an active research.

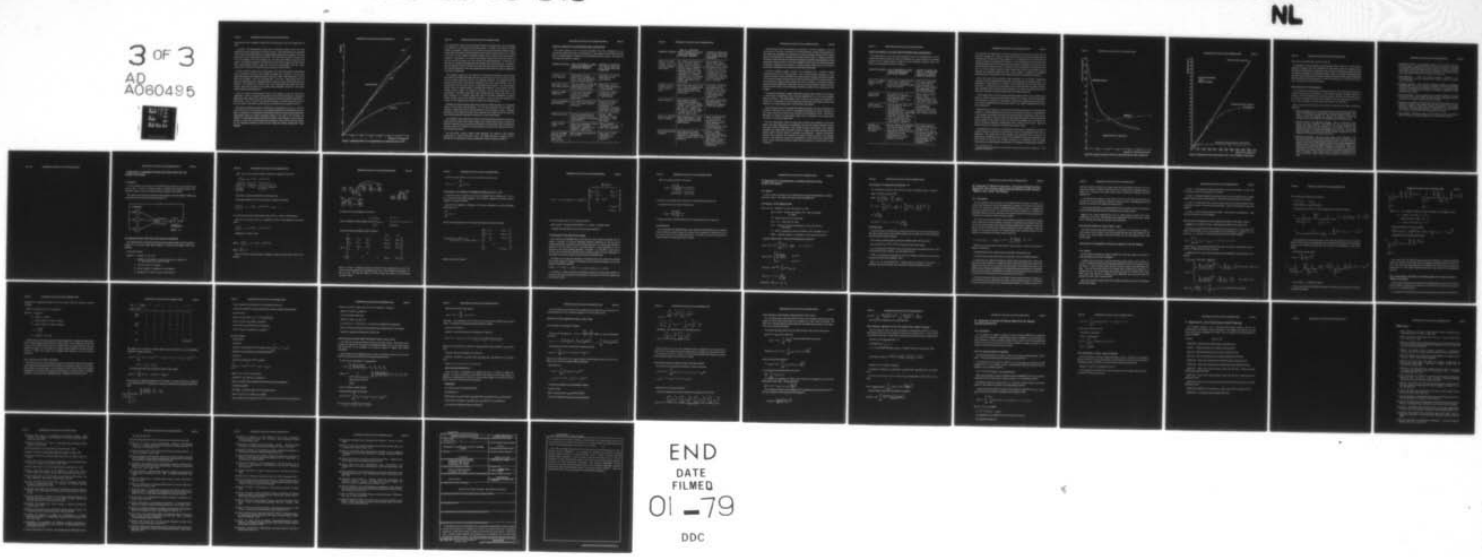
Four application programs, from different application areas, were used as the benchmark workload model for the performance investigation of the two structures. They were:

- Numerical application - Asynchronous iterative methods for the solution of Partial Differential Equations (PDE'S).
- Sorting application - Quicksort.
- Searching - Set Partitioning Integer Programming.
- AI, real time application - Harpy Speech Recognition System.

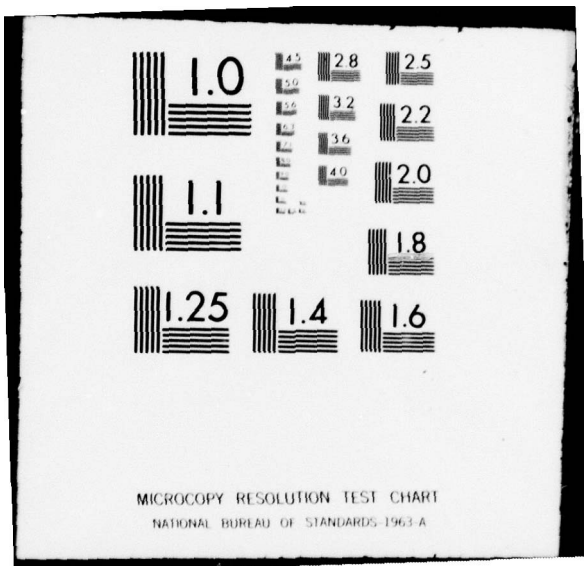
These applications were described in sections 2.3 and 4.3. A series of experiments and

AD-A060 495 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
PERFORMANCE EVALUATION OF MULTIPLE PROCESSOR SYSTEMS.(U)
AUG 78 L RASKIN
UNCLASSIFIED CMU-CS-78-141 F44620-73-C-0074
NL

3 OF 3
AD
A060495



END
DATE
FILMED
01-79
DDC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

measurements were conducted to determine the performance of this set of applications on Cm*.

In chapter 2 the decompositions issues and the measurement results of the first three applications on the multiprocessor structure, are given. The measurements were conducted on the initial Cm* system where up to eight computer-modules were available. The main observation was the significance of the *hit ratio* - the ratio of the processor's reference rate to its local memory to its total reference rate. The three level memory hierarchy in the system: local memory, memory in the same cluster and memory module in another cluster (with inter-reference times of about 3.5, 9.3 and 24 microseconds) make the system performance a function of the relative access frequencies to these modules.

Another important aspect was the measurement of the relative frequencies of referencing code, processor's stack area, local variables and *global read only* data. For the three application programs the relative frequencies were about: Code - 70% to 80%, Stack - 10% to 25%, Local variables - 4% to 7%, and Global Data - 1% to 10%. These figures show the significant improvement due to localizing code (i.e. replicating in each of the processor's local memory). The typically low reference rate to the shared global data means that these (usually large) data structures can be placed, without replication, anywhere in the system without excessive performance loss.

Figure 5.1 shows the measurement results of the speed-up factor for these three applications. While the Integer Programming and Partial Differential Equations applications approached linear speed-up, the Quick Sort application showed considerable degradation in performance. This is a result of not sufficient parallelism in the specific algorithm as more Pc's are added, as discussed in section 2.3, and not a limitation of the computer structure.

Quantitative measurement results of applications on multiprocessors are scarce. The main contributions of this part of the thesis are: **showing the possibility of solving efficiently several types of problems on a multiprocessor (multi-computer-module) structure, obtaining quantitative measurement results and various performance measures to be used in the derivation and validation of performance models (in particular the relative access frequency to various memory patterns, i.e. code, stack etc.). Demonstrating the possibility of using relatively simple techniques to get meaningful performance results, and the insight into problem decomposition issues obtained.**

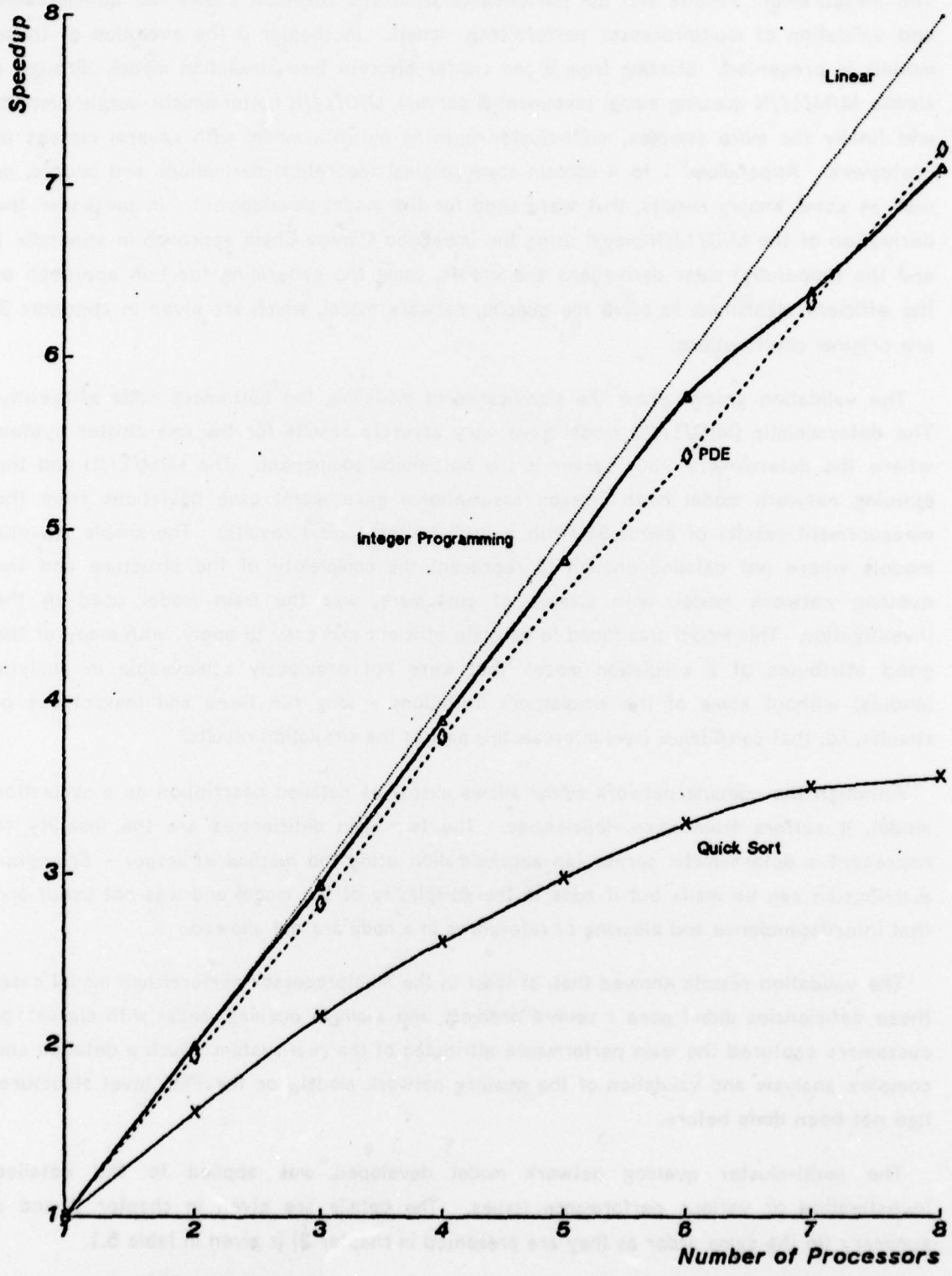


Fig.5.1: Speedup Factor of 3 Applications on multiprocessor Cm*.

The measurement results and the performance measures collected allows the development and validation of multiprocessor performance models. In chapter 3 the evolution of these models is presented. Starting from a one cluster discrete time simulation model, through a simple M/M/1//N queuing model (exponential server), M/D/1//N (deterministic server) model, and finally the more complex, multi-cluster, queuing network model with several classes of customers. Appendices 1 to 4 contain some original theoretical derivations and proofs, as well as some known results, that were used for the model development. In particular the derivation of the M/D/1//N model using the Imbedded Markov Chain approach in appendix 1 and the (hopefully) clear derivations and proofs, using the generating function approach of the efficient algorithms to solve the queuing network model, which are given in appendix 3, are original contributions.

The validation graphs show the significance of modelling the bottleneck node efficiently. The deterministic (M/D/1//N) model gave very accurate results for the one cluster system where the deterministic Kbus server is the bottleneck component. The M/M/1//N and the queuing network model (with Poisson assumptions) gave worst case deviations from the measurement results of about 8% (with a much better typical results). The simple analytic models where not detailed enough to represent the complexity of the structure and the queuing network model, with classes of customers, was the main model used in the investigation. This model was found to be quite efficient and easy to apply, with many of the good attributes of a simulation model (that were not previously achieveable in analytic models) without some of the simulation's limitations - long run times and inexactness of results, i.e. that *confidence level* intervals are part of the simulation results.

Although the queuing network model allows almost as detailed description as a simulation model, it suffers from some deficiencies. The two main deficiencies are the inability to represent a deterministic server (an approximation using the *method of stages - Erlangian distribution* can be made but it adds to the complexity of the model and was not used) and that interdependence and blocking of references in a node are not allowed.

The validation results showed that, at least in the multiprocessor performance model case, these deficiencies didn't pose a severe problem, and a single queuing model with classes of customers captured the main performance attributes of the real system. Such a detailed and complex analysis and validation of the queuing network model, for the PMS level structure, had not been done before.

The multi-cluster queuing network model developed was applied to the detailed investigation of various performance issues. The details are given in chapter 3 and a summary (in the same order as they are presented in chapter 3) is given in table 5.1.

TABLE 5.1: SUMMARY OF MULTIPROCESSOR'S MODEL INVESTIGATIONS

This table contains a list of the main parameters that were investigated by the performance model, their effect on performance of a large system, and an example, derived from the relevant figures in chapter 3, on the parameter's saturation values. For details see the appropriate section in chapter 3.

PARAMETER CHECKED	MAIN ATTRIBUTES OF A LARGE SYSTEM AND PERFORMANCE BOTTLENECK COMPONENT.	EXAMPLE OF SATURATION (HIT RATIO = 0.9, 50 Cms, UNLESS SPECIFIED OTHERWISE).
Hit Ratio (to local memory).	Performance is very sensitive to changes. Hit ratio should be maximized. Kbus is the bottleneck.	5% change from 0.9 to 0.95 doubles the performance.
Share Ratio (to one memory module)	High Share Ratio causes the shared memory to be the performance bottleneck.	Share Ratio of 0.5 causes 20% degradation in performance.
Number of Kmap "contexts".	8 "contexts" are about right. More contexts would not increase performance	4 "contexts" cause 20% degradation in performance.
Faster processors and memories.	Give better performance up to a limit when the Kbus saturates.	Processor X3 faster, 12 Cms gives 60% improvement in performance. processor -memory pair X3 faster gives 95% improvement.
Changing Map bus.	Eliminating the Kbus and short arbitration time in the bus improves performance considerably.	No Kbus - 90% improvement. Faster arbitration - 110% improvement.
Changing Kmap clock.	Sensitive to changes in Kbus clock (the bottleneck component). Insensitive to changes in Pmap clock.	25% performance improvement for 25% Kbus clock decrease.
High performance system (fast X3 Pc- memory, no Kmap, fast arbitration).	Much higher performance can be achieved.	More than 5 times improvement in performance is possible.

TABLE 5.1 (CONTINUED)

PARAMETER CHECKED	MAIN ATTRIBUTES OF A LARGE SYSTEM AND PERFORMANCE BOTTLENECK COMPONENT.	EXAMPLE OF SATURATION (HIT RATIO = 0.9, 50 Cms, UNLESS SPECIFIED OTHERWISE).
Comparing single cluster to multi-cluster structure, checking memory hierarchy.	For random distribution of data, a single cluster is better than a 4 cluster system (inter-cluster bus is the bottleneck). better decompositions with lower inter-cluster rates (less than 50% of all external references for a 4 clusters system) multiclusters are better (the Kbus is not saturated).	One cluster has a better performance than a 4 cluster, 50 Cm system, by a factor of 1.3. For inter - cluster ratio of 25%, 4 clusters are 2.3 times better than for a random distribution (with inter-cluster ratio of 0.75).
Changing inter - cluster parallel bus to serial line.	Relatively fast serial line still causes a considerable performance degradation. Inter-cluster bus is the bottleneck.	2 clusters, 10 Mbits per second serial line bus with inter-cluster ratio of 0.5 causes more than 3 times performance degradation.
Cache memories.	Cache memories cause performance degradation compared to current system (higher total "miss ratio"). but is still feasible (in particular if faster cache is used) and cut the total required memory size.	2K cache causes about 35% performance degradation compared to current system.
Effect of one I/O block transfer.	Small degradation of total system performance. Large system slows the I/O rate considerably.	about 5% degradation in total system performance. 5 to 6 times degradation in I/O rate with increased system size from 1 to 50 Cms.
Performance/memory tradeoff.	Very sensitive to code localization. Relatively insensitive to placement of read only global data in the structure.	22 Cms, Integer Prog. application: localizing code gives about 8 times improvement. Global data localization results in only 2.5% improvement.

These examples show the practicality and usefulness of the tools developed. The main contributions of this part of the thesis are: the theoretical work involved in the development of the models, the investigation and comparisons of several performance models and their relative merits and deficiencies, the validation of the queuing network model as a capable and efficient performance model, the ease and practicality of applying this model for the investigation of many performance issues (as seen in the various examples), and the insight that these performance investigation examples give the architect of a multiple computer-module system on the effect of performance modifications and changes on the total system's performance.

The local computer network structure was investigated next. Firmware changes (microprogramming of the Kmap processor in the structure) allowed emulation of a local computer network structure (Cm-Net), that included statistical package and enabled changes in the *network transport* communication rate (that are difficult to obtain on a real network). A *process interface software support* package was written to simulate a performance oriented operating system supporting an application on top of a the local computer network structure. Several protocol options (e.g. acknowledge mechanisms, broadcast) were supported by this package.

Two application programs: Integer Programming and Harpy where chosen as two *extremes* in inter-process communication rates, and were modified to run on the local computer network structure. The decomposition issues involved in that process are discussed in section 4.3 - the emphasis was on minimizing the communication rate between processes and replication of all the *read-only* data in the processor's local memory.

The performance result of the Integer Programming application, with matrix replication, was excellent, while Harpy and a version of the Integer Programming with matrix centralized at the *master* process local memory, had a relatively poor performance results - showing the advantage of utilizing the large address space with low overhead, of the multiprocessor structure, for this class of applications.

From the measurement results we were able to identify nine parameters as the main factors that affect performance of an application using the local computer network structure (e.g. *slave* process communication rates, *network transport* communication rate, message overhead, etc.). Using these parameters as input a simple, three node, queuing network analytic model with four classes of customers was constructed. Validation experiments showed that it represent the application and network structure behavior with adequate accuracy. The model was used for the investigation of several local network performance issues. These results are discussed is section 4.4 and a summary of the main points is given in table 5.2.

TABLE 5.2: SUMMARY OF THE LOCAL COMPUTER NETWORK MODEL INVESTIGATIONS

This table contains a list of the main parameters that were investigated by the local network performance model, their effect on performance of a large system and an example, derived from the relevant figures in chapter 4, on the parameter's saturation values. For details see the appropriate section in chapter 4.

PARAMETER CHECKED	MAIN ATTRIBUTES OF A LARGE SYSTEM AND PERFORMANCE BOTTLENECK COMPONENT.	EXAMPLE OF SATURATION (50 Cms, INTEGER PROG. APPLICATION, UNLESS SPECIFIED OTHERWISE).
Network transport communications rate.	sensitive to communications rate up to a saturation point above which performance does not improve.	2.5 times improvement for communication rate increase from 0.16 to 1.3 Mbits/sec. No improvement from 10 to 50 Mbits/sec.
Acknowledge mechanism.	Not sensitive for low message rate. Sensitive in high message rates. Software overhead is the bottleneck.	8 Cms, Centralizing matrix (high communications rate). 1.5 times better with "hardware" acknowledge.
Comparison to multiprocessors.	Sharing address space with low overhead in multiprocessors is considerably better for large systems (even for low message rate applications). Not important for small systems with low communications rates. Larger memory size requirement in networks.	Performance of multiprocessor version is more than twice better than the local network version.
Classes of applications suitable for a structure.	Optimal number of processors or network communications rate can be found for an application. On the contrary, classes of suitable applications can be determined for a given network structure.	For communications rate of 1.3 Mbits/sec., and 50% return criteria (i.e. 50% of the maximum capability of the processor is utilized) give that 23 processors are the optimum number of processors.

The following two figures point to the main performance issues of the structure and are therefore repeated here. Figure 5.2 shows the execution time of both the network and the multiprocessor versions of the Harpy application. It is interesting to see the effect of the tradeoffs involved in the decomposition process: In the multiprocessor the trend was toward maximizing performance with a large number of processors participating in the task (and thus the performance with small number of processors was not emphasized). We realized a priori that in the large network version the performance would be poor and therefore tried to maximize the absolute performance with a small number of processors, i.e. different decompositions of the algorithm result in better performance in the different structures.

Figure 5.3 show the performance model predictions for the Integer Programming application (without considering the perbutations in the algorithms as mentioned before). The degradation in performance in the network - matrix replicated case, with large number of processors - is mainly due to the initialization time for the replication of the matrix and other *read only global* data in all the processor's memories. The matrix centralized version suffers from early saturation of the *master* process due to the high message rate.

These figures point to a few important performance aspects; The first, that multiprocessors provide a saving in memory size requirement compared to a network - due to the possibility of localizing the *read only data* in one memory module without significant performance loss - as is seen by comparing the multiprocessor results to the Integer Programming with matrix centralized graph.

Second, and most important, is the observation that the performance of a local computer network is a multi-variable function, with three main variables that affect the performance[1] : The message rate between processors, the *network transport* communications rate, and the number of processors. One can envision the total system performance as a three dimensional space problem with performance, number of processors and message rate as the three axes and the *transport mechanism* communications rate as surfaces in that space. Figures in section 4.5 show cuts through this three dimensional space surfaces.

Investigating this space led to the observations that classes of applications are able to efficiently perform on a given structure, and that a structure can be designed to best suit an application. Best suited require more variables to be considered, such as *cost, ease of programming, availability, etc.*, we assume that the implementor is able to consider these. By fixing three of the four main variables, according to an application and the design requirements, an optimal design can be carried out. An example of chosing optimal number of

---

[1] We have assumed that the value of the other parameters are either relatively constant in an application or of less significance in a typical case.

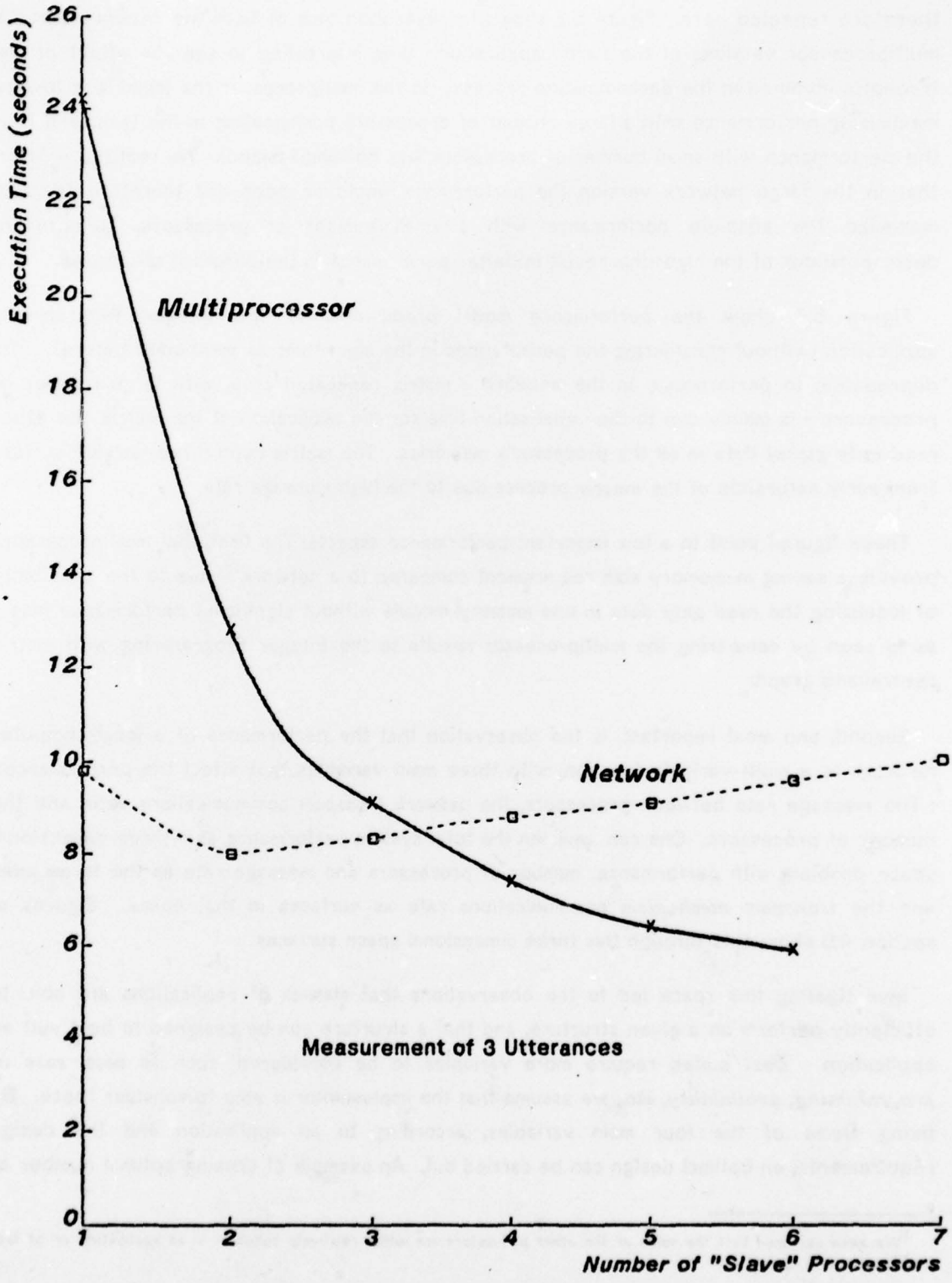


Fig.5.2: Harpy, Execution Time of a Multiprocessor and a Network

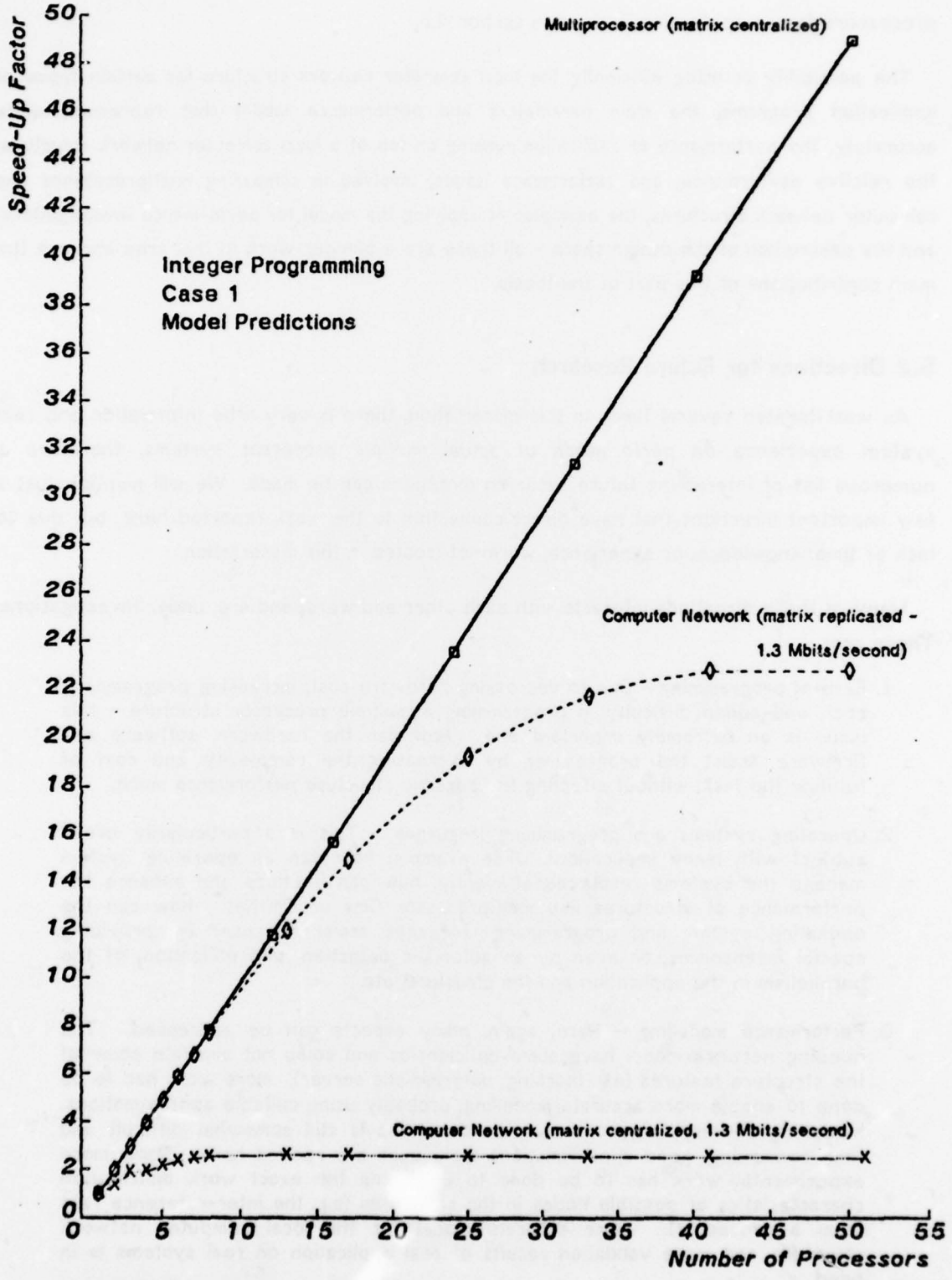


Fig.5.3: Speedup of Multiprocessors and Local Computer Networks

processors for an application is given in section 4.5.

The possibility of using efficiently the local computer network structure for certain types of application programs, the main parameters and performance model that represent, quite accurately, the performance of application running on top of a local computer network structure, the relative performance, and performance issues, involved in comparing multiprocessors and computer network structures, the examples of applying the model for performance investigations, and the description of the design space - all these are a pioneer work in this area and are the main contributions of this part of the thesis.

5.2 Directions for Future Research

As was iterated several times in this dissertation, there is very little information and real system experience on performance of actual multiple processor systems, therefore a numerous list of interesting future research directions can be made. We will mention just a few important directions that have direct connection to the work reported here, but due to lack of time, knowledge, or experience, were not treated in this dissertation.

Many of these directions interacts with each other and were, and are, under investigations. These are:

1. Ease of programming - Due to decreasing hardware cost, increasing programming cost, and added difficulty in programming a multiple processor structure - this issue is an extremely important one. How can the hardware, software and firmware assist the programmer by decreasing the complexity and cost of fulfiling the task, without affecting the possible structure performance much.

2. Operating systems and programming languages - This is a particularily broad subject with many implications. For example; how can an operating system manage the systems resources efficiently, how can it utilize and enhance the performance of structures like multiprocessor Cm* or Cm-Net. How can the operating system and programming languages assist the user by providing special mechanisms, or even by an automatic detection, and utilization, of the parallelism in the application and the structure, etc.

3. Performance modelling - Here, again, many aspects can be addressed. The queuing network model have some deficiencies and could not simulate some of the structure features (e.g. blocking, deterministic server). more work had to be done to enable more accurate modelling, probably using suitable approximations. Modelling of very large and complex structures is still somewhat difficult and time consuming, good approximations techniques are needed here. Some more experimental work has to be done to determine the exact work distribution characteristics of possible nodes in the structures (e.g. the inter-reference rate from a processor). More accurate model for the local computer network structure and more validation results of real application on real systems is in need.

4. **Input/output aspects** - The input/output aspects of the structures was only scantly touched. The benchmark workload, that we have used, was mainly made of compute-bound applications that did not explore that subject. A detailed investigation, involving both measurement and modelling, is needed to cover this issue. The possible incorporation of various I/O devices in the systems (drums, disks, CCD, bubble memories etc.) complicates this issue further.

5. **Multi-programming** - Using the multiple processor structures in an multiprogramming, multi user, environment has different aspects and demand specific treatment.

6. **Integrated modelling - Cost, reliability, availability, protection and application modelling issues** - All these are examples of important and difficult issues which are part of the performance and cost/performance aspects. An integrated model, probably at the PMS level for the structure, is necessary to enable an optimal cost/performance design.

7. **Future Cm* system** - This work touched on many aspects and effects of changes in the current Cm* system. Based on these, and other, investigations the next generation of a computer-module system can be designed.

8. **Experience and validation of a large system** - More work and experience of a real applications utilizing a large multiple processor system (with many processing elements) is needed to validate many performance assumptions. The currently built 50 Cms Cm* system holds promises in that direction. The small address space problem should be also further investigated in that context.

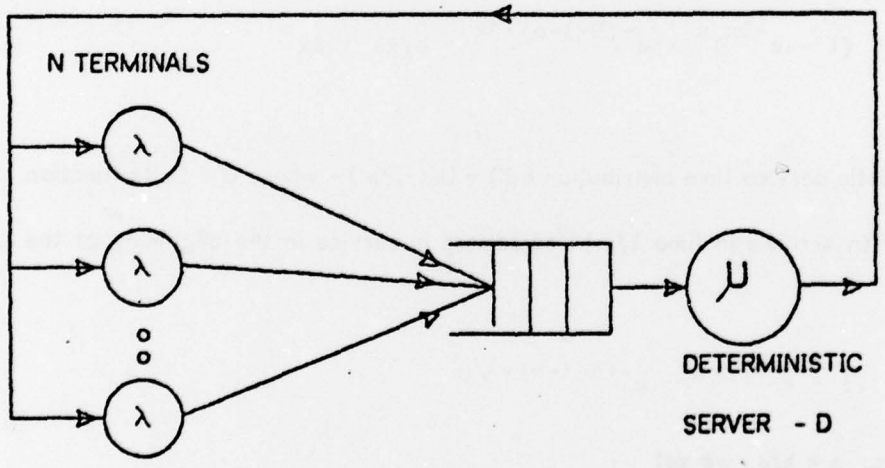
9. **Local networks** - Performance issues of local networks where not addressed before. We hope to see more research activity in that area in the future.

I. Appendix 1: Imbedded Markov Chain Derivation for the M/D/1//N Queue

I.1 General

In this appendix the derivation of the M/D/1//N queue (Poisson arrivals, Deterministic service time, 1 server, N customers) using a straightforward Imbedded Markov Chain approach, is given. We were not able to find this derivation in the literature.

The results were found to be identical to those obtained by Jaiswal [JAISWAL 1968] using the Supplementary Variable technique (see appendix 2).



I.2 Imbedded Markov Chain States and Transition Probabilities

The imbedded Markov chain epochs will be the service departure times, with the number of customers in the service center (either queued or in service) as the Markov chain state variables.

a. Derivation of A(n,i)

Assume: n - Number of arrivals

i - Number of customers in service following a departure (in beginning of new service time).

λ - Arrival rate of a source.

N - Total number of customers in the system.

μ - Deterministic service rate of the server.

Then: A(n,i) = Pr (n arrivals in arbitrary time t given i customers in service) =

$$\underbrace{\binom{N-i}{n}}_{\text{chose n out of (N-i) customers available}} \cdot \underbrace{[1 - e^{-\lambda t}]^n}_{\text{n arrivals in time t}} \cdot \underbrace{e^{-(N-i-n)\cdot\lambda t}}_{\text{(N-i-n) not arrived in time t (remained in source}}$$

Given B(X) - service time distribution of the server, then

Pr(average number of arrivals in service time | i customers in service)=

$$\int_0^\infty \binom{N-i}{n} \cdot (1 - e^{-\lambda x})^n \cdot e^{-(N-i-n)\cdot\lambda x} \cdot b(x) \cdot dx$$

For deterministic service time distribution b(X) = U(X-1/μ) - where U = Delta function.

A(n,i) = Pr (n arrivals in time 1/μ | i customers in service in the beginning of the service time) =

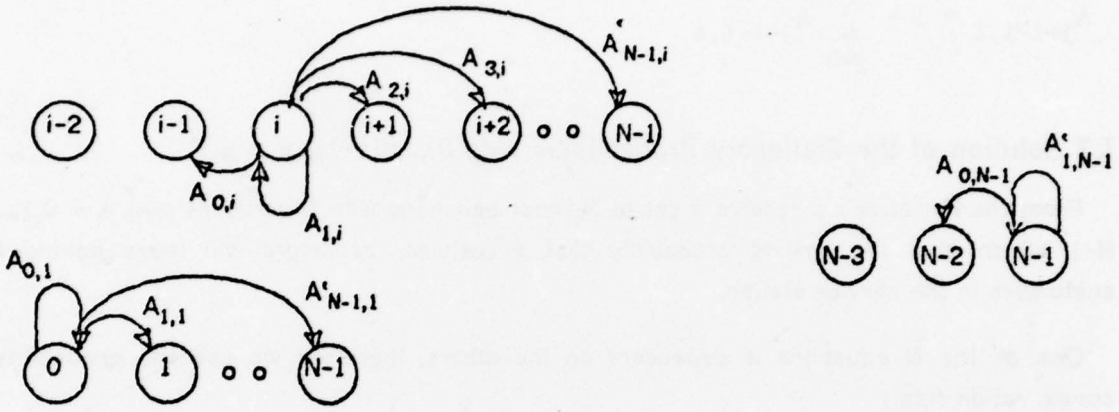
$$\binom{N-i}{n} \cdot (1 - e^{-\lambda/\mu})^n \cdot e^{-(N-i-n)\cdot\lambda/\mu}$$

Designating: $\rho = \lambda/\mu$, we get

$$A(n,i) = \binom{N-i}{n} \cdot (1 - e^{-\rho})^n \cdot e^{-(N-i-n)\cdot\rho}$$

Where $\binom{m}{0} = 1$

Given the above, the state transition probability diagram (from one markov state to the other) is:



By observing the state diagram we can write:

$$P(i,j) = \Pr(\text{going from state variable } i \text{ to } j) = \begin{cases} A_{j,1} & \text{if } i = 0 \\ A_{j+1-i,i} & \text{if } i \neq 0 \wedge N > j+1-i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The state transition probability matrix is therefore:

$$\mathbb{P} = \begin{array}{c} \\ 0 \\ 0 \\ \\ \\ N-1 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & & & & N-2 & N-1 \end{array} \\ \begin{bmatrix} A_{0,1} & A_{1,1} & A_{2,1} & & A_{N-2,1} & A'_{N-1,1} \\ A_{0,1} & A_{1,1} & A_{2,1} & & A_{N-2,1} & A'_{N-1,1} \\ 0 & A_{0,2} & A_{1,2} & & A_{N-3,2} & A'_{N-2,2} \\ 0 & 0 & & & & \\ & & 0 & & & \\ 0 & 0 & 0 & & A_{0,N-1} & A'_{1,N-1} \end{bmatrix} \end{array}$$

Notes: (a) P(0,j) - probability of going from state 0 to state j depends on an arrival. We start to count the service time only following that arrival - which explains the subscript 1 in the first row. The idle time before an arrival is taken into account in the (following) correction action.

(b) The last column (N-1) is, in fact, a result of the conservation rule:

$$A_{j+1-i,i} = 1 - \sum_{j=0}^{N-2} A_{j+1-i,i}$$

I.3 Solution of the Stationary Probabilities p=(p(0),..,p(n-1)) = p x P

From the P matrix we receive a set of N linear equations with N variables p(k), k = 0,1,..., N-1 which give the limiting probability that a customer departure will leave behind k customers in the service station.

One of the N equations is dependent on the others, therefore we use the probability conservation rule :

$$\sum_{i=0}^{N-1} P(k) = 1$$

The set of linear equations (with the conservation equation as the last column) is:

$$[p(0),p(1),\ldots,p(N-2),1] = [p(0),p(1),\ldots,p(N-2),p(N-1)] \cdot \begin{vmatrix} A_{0,1} & A_{1,1} & \cdots & A_{N-2,1} & 1 \\ A_{0,1} & A_{1,1} & & A_{N-2,1} & 1 \\ 0 & A_{3,2} & & A_{N-3,2} & 1 \\ 0 & 0 & & & 1 \\ & & 0 & & \\ 0 & & 0 & A_{0,N-1} & 1 \end{vmatrix}$$

Another way to write this set:

$$[0,0,\ldots,0,1] = p(0),p(1),\ldots,p(M-1)] \cdot \overbrace{\begin{bmatrix} A_{0,1}^{-1} & A_{1,1} & & & A_{N-2,1} & 1 \\ A_{0,1} & A_{1,1}^{-1} & & & A_{N-2,1} & 1 \\ 0 & A_{0,2} & A_{1,2}^{-1} & & & 1 \\ & 0 & & & & \\ & & & & A_{1,N-2}^{-1} & 1 \\ 0 & & & 0 & A_{0,N-2} & 1 \end{bmatrix}}^{\mathbb{P}_1 \text{ matrix}}$$

If we invert the $\mathbb{P}_1$ matrix - $\mathbb{P}_1^{-1}$ then the solution is:

$[0,0,\ldots,0,1]\mathbb{P}_1^{-1} = [p(0),p(1),\ldots p(N-1)]\mathbb{P}_1\mathbb{P}_1^{-1} = p\ I$; Where I = Identity matrix.

Therefore p= (p(0),...p(N-1)) = last row of the inverse matrix $\mathbb{P}_1^{-1}$.

I.4 Correction to the Idle Server Problem

The results of the Imbedded Markov chain give the correct limiting probabilities p(k) for finding k customers in the server immediately following a departure, but not the correct limiting probabilities in an arbitrary time t between departures. For example, p(N) following a departure is 0, but there can be a time (and therefore a finite probability) to find N customers in the system between departures. The M/D/1 - or even M/G/1- results hold in all times due to the infinite population and the constant average arrival rate independent of the number in the server, which is not true in the finite population case.

Fortunately, in our case, the probability of finding the server idle is sufficient to calculate the average number of customers in the server, and the required throughput, as we have derived (using balance equation)-

$(N-\bar{p}_k) \cdot \lambda = (1-p_i) \cdot \mu \quad \bar{p}_k = N - (1-p_i)/\rho$; where p_i = p (idle)

To derive p_i = p(idle) we observe that following a departure with only zero customers in the server (all N are in the source) the probability of no arrivals is $e^{-\lambda t}$, and the average time to an arrival is $1/N\lambda$.

Thus we can derive p(idle) by the equation:

$$p_{idle} = \frac{p(0)\cdot\frac{1}{N\lambda}}{\underbrace{p(0)\left(\frac{1}{N\lambda}+\frac{1}{\mu}\right)}_{p(k)=p(0)} + \underbrace{(1-p(0))\cdot\frac{1}{\mu}}_{p(k)\neq p(0)}} =$$

= idle time in one average inter-departure time / average inter-departure time.

Or alternatively, we can directly interpret it as:

$$p_{idle} = \frac{p(0)\cdot\frac{1}{N\cdot\lambda}}{p(0)\cdot\frac{1}{N\cdot\lambda}+\frac{1}{\mu}} =$$

= average idle time in one inter-departure time / average time between departures.

I.5 Conclusion

In this appendix the Imbedded Markov chain solution to the M/D/1//N queue is derived. These results were used (in chapter 3) to compare the throughput of a deterministic server to a Poisson server in the multiprocessor model. The results were found to be identical to those obtained by Jaiswal using another technique.

II. Appendix 2: Supplementary Variables Solution to the M/D/1//N Queue

II.1 General

In this appendix the solution to the M/D/1//N queue, using the Supplementary Variables technique, is given. The complete derivation is given in [JAISWAL 68].

II.2 Solution of the M/G/1//N Case

Notation: $\hat{e}$ - Probability that the server is idle.

U_c or $(1-\hat{e})$ - server utilization (Pr. that the server is busy).

$\bar{b}$ - mean busy period of the server.

$1/\mu$ or ν - mean service time.

$1/\lambda$ - mean of the source processing time distribution (Poisson).

X - server processing time with density $f(X)$ and mean $1/\mu = \nu$.

$E(\hat{m})$ - expected number of customers in the service station.

Using the notation above - the main results obtained by Jaiswal are:

$$E(b) = \bar{b} = (1/\mu) \sum_{i=0}^{N-1} \binom{N-1}{i} \cdot \frac{1}{\varphi(i)} \qquad \text{if } \nu = 1/\mu < \infty$$

$$\text{Where } \varphi(m) = \begin{cases} \prod_{i=1}^{m} \frac{\bar{S}(i\lambda)}{1-\bar{S}(i\lambda)} & \text{if } m \neq 0 \\ = 1 & \text{if } m = 0 \end{cases}$$

$$\text{and } \bar{S}(s) = E[e^{-sX}] = \int_0^{\infty} e^{-sX} f(X)\, dX$$

$$\text{then } U_c = 1 - \hat{e} = \frac{\bar{b}}{\bar{b} + \frac{1}{N\lambda}}$$

$$\text{and finally } E(\hat{m}) = N - \frac{\mu}{\lambda} \cdot U_c$$

II.3 Solution for Deterministic Distribution - D

For deterministic distribution $f(X) = U(X-1/\mu)$ - where U is the Delta function. Therefore:

$$S(i\lambda) = e^{-i\lambda X}\, U(X-1/\mu)\,dX = e^{-i\lambda/\mu}$$

$$\varphi(m) = \prod_{i=1}^{m} \frac{e^{-i\lambda/\mu}}{1-e^{-i\lambda/\mu}} = \prod_{i=1}^{m} \frac{1}{e^{i\lambda/\mu}-1}$$

$$\bar{b} = (1/\mu) \cdot \sum_{i=0}^{N-1} \binom{N-1}{i} \cdot \frac{1}{\varphi(i)} = \frac{1}{\mu} \cdot \sum_{i=0}^{N-1} \binom{N-1}{i} \cdot \prod_{m=1}^{i} \left(e^{m \cdot \frac{\lambda}{\mu}} - 1 \right)$$

$$U_c = \frac{\bar{b}}{\bar{b} + \frac{1}{N \cdot \lambda}}$$

$$\text{and } E(\hat{m}) = N - \frac{\mu}{\lambda} \cdot U_c = N - \frac{\mu}{\lambda} \cdot \frac{\bar{b}}{\bar{b} + \frac{1}{N \cdot \lambda}}$$

II.4 Discussion

As was shown by Price [PRICE 76] a constant (deterministic) service time distribution gives the upper bound for the server utilization (better than any other service time distribution). This can be easily shown as follows:

From Jensen's inequality $E[g(x)] \geq g[E(x)]$ (see PARZEN 1960 P. 434) we receive:

$S(s) = E[\, e^{-sX}\,] \geq e^{-s\,E[x]} = e^{-s/\mu}$ = Transform of deterministic function.

U_c is increasing with b (length of server busy period) and b is decreasing with $\phi(m)$. $\phi(m)$ is increasing with S(s), Therefore - U_c is decreasing with S(s).

$e^{-s/\mu}$ is the minimum S(s) possible (for same $1/\mu$), therefore constant service time gives the upper bound on server utilization. q.e.d.

$E(m) = N - U_c\ \mu/\lambda$, therefore E(m) - average number of customers in the server - is minimized when U_c is maximized and is minimum for deterministic distribution server.

III. Appendix 3: Efficient Algorithms, Using Generating Function Approach, For Solving Closed Queuing Networks with Several "Global" and "Local" Job Classes.

III.1 Introduction

In this appendix proofs are given for the algorithms that were used in chapter 3 to evaluate the performance of Cm*-like multiprocessor architecture. This work is an extension to the work reported by Williams & Bhandiwad [Williams 76] that used a straightforward Generating Function approach. This work rely on the results of Baskett et al. [Baskett 75] (the BCMP theorem or model). See chapter 3 for general discussion of the queuing networks, in particular references to similar work done by others.

For type 2 P.S. (Processor Sharing) servers, Williams and Bhandiwad (WB model) showed, using a Generating Function approach, an algorithm to compute the normalization constant C for the case of two "general" job classes (i.e. closed queueing network where the sum of the jobs in the first and second class is constant and equal to m and n respectively). They also showed (as can be seen directly from the BCMP theorem) that the general equation for the steady state probabilities with type 2 servers, is given by:

$$\Pr\ (m_1,n_1;m_2,n_2;\ldots;m_M,n_M] = (1/C) \cdot \prod_{i=1}^{M} \frac{(m_i+n_i)!}{m_i!\cdot n_i!} \cdot X^{m_i} \cdot Y^{n_i}$$

Where m_i,n_i are the number of jobs of class 1 and 2 in server i, and M is the number of nodes in the networks.

The modifications to their results, which are presented in this appendix, are:

a. Showing that the same results hold for servers type 1 and 4 (in the BCMP theorem).

b. Introducing the "global" and "local" job classes notion and proving that if there are "local" job classes transitions in a "global" job class (i.e. jobs that belong to one "global" class change their "local" class in a node, but the total number of the "global" class jobs in the closed network remains unchanged) we can replace the detailed enumeration of the individual "local" job classes terms in the algorithm by a single term, one equivalent "job", with the sum of the ρ's (X's in the WB notation) as the equivalent load of the server.

This result can also be used in the one "global" job class queueing case, and results in a very significant saving in computation of relatively complex queueing networks. For example, in cases where each job (e.g. a job in multiprogrammed computer or a memory request in an

hierarchical memory structure) can change classes while traversing the network, and visit a node several times as different "local" class (see for example the Cm* queueing network in chapter 3). This result is also given (without proof or reference) by Gelenbe and Muntz [Gelenbe 76].

c. The WB recursive algorithm is given only for the case of two job classes and does not provide with the exact algorithm to initialize and apply the results. The detailed proof of the algorithm for more than two "global" job classes is given.

d. Proving the algorithm for incorporating type 3 L.S. node (Infinite servers) in the network and showing the significant reduction in computation for this type of server.

Note: all the results presented here are for closed queueing network with queue independent service rates. Generalization to queue dependent servers can be done in similar fashion (although they make the algorithms more cumbersome).

III.2 General Solution for Servers Type 1 and 4

The proof that the same general solution, as was given in the introduction, applies to server types 1 and 4 is trivial - as all the three types has exactly the same equation for the product form solution ([Baskett 75]), which implies that the same general steady state solution holds.

III.3 Proof of the Algorithms for the Case of Several "Local" Job Classes.

III.3.1 General

In this section we define the notion of "global" and "local" job classes for the type of closed queueing network under investigation.

A "global" job class contains n jobs (n tokens) that traverse some, or all, the nodes in a network. A "global" job has "local" job classes associated with it, and it can change its "local" job class in a node to another "local" job class - but still remain in the same "global" job class. The sum of the jobs (tokens) in that "global" class is constant and is equal to n. A "global" job may visit the same node several times, each time as a different "local" class. Each "local" class may have a different service requirement from the server (with the exception of server type 1 as shown below).

Given the restrictions stated above, and the BCMP restrictions, we allow the following service center types:

a. Type 1 - FCFS discipline, negative exponential service time distribution. All jobs have the same mean service time request from the server.

b. Type 2 - Processor Sharing (P.S.) discipline, Coxian (i.e. any distribution with a rational Laplace transform) service time distribution. Jobs may have different service time request.

c. Type 3 - Infinite Server (I.S.). At least as many servers as the maximum number of jobs in that service station. Coxian service time distribution. Jobs may have different service time requests.

d. Type 4 - LCFS, Preemptive Resume discipline. Coxian service time distribution. Jobs may have different service time requests.

III.3.2 Proof of the "Equivalence" of a One "Global" Job to Several "Local" Jobs in a Server

Notation (for one "global" job class - following partially WB and Muntz [Muntz 74]):

Let $(s_1,s_2,\ldots s_M)$ be the state of the network with M servers, where $s_i = (n_{i,1},\ldots,n_{i,R})$ is the state of service center i and $n_{i,r}$ is the number of jobs of "local" class r in server i; $1 \le r \le R$. Let $\lambda_{i,r}$, $1 \le i \le M$ and $1 \le r \le R$ be the solution to the set of linear equations:

$$\lambda_{i,r} = \sum_{j=1}^{M} \sum_{k=1}^{R} \lambda_{j,k} \cdot p_{j,k;i,r}, \quad 1 \le i \le M,\ 1 \le r \le R$$

Where $p_{j,k;i,r}$ is the branching probability that a job of local class k, that finishes service at server j, will seek service at server i as class r.

In [Baskett 75] it is shown that the equilibrium state probabilities have the product form solution:

$$\Pr(s_1,\ldots,s_M) = 1/C\ F_1(s_1) \ldots F_M(s_M) \text{ and}$$

$$F_i(s_i) = \begin{cases} n_i! \prod_{r=1}^{R} \frac{1}{n_{i,r}!}\left(\frac{\lambda_{i,r}}{\mu_{i,r}}\right)^{n_{i,r}} = n_i! \prod_{r=1}^{R} \frac{1}{n_{ir}!} \cdot \rho_{i,r}^{n_{i,r}} & \text{For server types 1,2,4} \\ \prod_{r=1}^{R} \frac{1}{n_{ir}!}\left(\frac{\lambda_{i,r}}{\mu_{i,r}}\right)^{n_{i,r}} = \prod_{r=1}^{R} \frac{1}{n_{i,r}!} \cdot \rho_{i,r}^{n_{i,r}} & \text{For server type 3} \end{cases}$$

Where $\rho_{i,r} \triangleq \frac{\lambda_{i,r}}{\mu_{i,r}}$; $n_i = \sum_{r=1}^{R} n_{i,r}$, C is the normalization constant.

The normalization constant can be found by:

$$C = \sum F_1(s_1) \ \ldots\ F_M(s_M)$$

over all states $s_1, \ldots s_M$ of the network.

a. For servers type 1,2 and 4 , with n total jobs and n_i "local" jobs in server i:

$$C = \sum_{\substack{\text{over all possible} \\ \text{states } \sum_i n_i = n}} \prod_{i=1}^{M} \left(\sum_{\substack{r \text{ different classes} \\ \sum_r n_{i,r} = n_i}} n_{i!} \cdot \frac{1}{n_{i,1!} \cdots n_{i,R!}} \cdot \rho_{i,1}^{n_1} \cdots \rho_{i,R}^{n_R} \right)$$

$$= \sum_{\text{over all possible states with } \sum_i n_i = n} \prod_{i=1}^{M} (\rho_{i,1} + \rho_{i,2} + \ldots + \rho_{i,R})^{n_i}$$

The interpretation of these results allows us to replace the summation over all the possible different "local" states by one "equivalent" job in the node, with the number of the jobs being the sum of the different "local" jobs and with ρ_i the sum of the different $\rho_{i,r}$ in the server (node).

b. Similarly for server type 3:

$$F_i(s_i) = \prod_{r=1}^{R} \frac{1}{n_{i,r!}} \rho_{i,r}^{n_{i,r}}$$

$$C = \sum_{\substack{\text{over all} \\ \text{states } \sum_i n_i = n}} \prod_{i=1}^{M} \left(\sum_{\substack{\text{all states} \\ \sum_r n_{i,r} = n_i}} \frac{1}{n_{i,r!}} \cdot \rho_{i,r}^{n_{i,r}} \right) = \sum_{\substack{\text{all states} \\ \sum_i n_i = n}} \prod_{i=1}^{M} \frac{1}{n_{i,r!}} (\rho_{i,1} + \ldots + \rho_{i,R})^{n_i}$$

c. For several - L - "global" job classes:

for the case of several (L) "global" job classes, and a mixture of server types, the equation to compute C becomes:

$$C = \sum_{\substack{\text{all states} \\ \sum_i n_i^e = A \\ \sum_i n_i^e = B}} \prod_{i=1}^{M} \prod_{e=1}^{L} \prod_{r=1}^{R_e} F_i(n_{i,r}^e) = \sum_{\substack{\text{all states} \\ \sum_i n_i^e = A \\ \sum_i n_i^e = B}} \prod_{i=1}^{M} \prod_{e=1}^{L} \left(\sum_{\substack{\text{all states} \\ \sum_r n_{i,r}^e = n_i}} \prod_{r=1}^{R_e} F_i(n_{i,r}^e) \right)$$

Where : A , B - constant number of jobs of "global" job classes 1 and L.

e - "global" job class $L \geq e \geq 1$

i - node (server) number $M \geq i \geq 1$

r - "local" job class $R_e \geq r \geq 1$

Denoting the results in a. and b. above by:

$F_i^{e*}(s_i) = (\rho_{i,1}^e + \ldots + \rho_{i,R}^e)^{n_i^e}$, and $F_i^{e*}(s_i) = \frac{1}{n_i^e!} (\rho_{i,1} + \ldots + \rho_{i,R})^{n_i^e}$

respectively, we can see that C factors into:

$$C = \sum_{\substack{\text{all states such that} \\ \sum_i n_i^e = A \\ \sum_i n_i^e = B}} \prod_{i=1}^{M} \prod_{e=1}^{L} F_i^{e*}(s_i)$$

This result shows the significant saving achieved by being able to replace the different "local" job classes by an equivalent job, with parameters replaced by the sum of the ρ's of the various "local" jobs that belong to the same "global" class in the server, thus saving the enumeration of all the possible different "local" jobs in the solution of the normalization constant.

III.4 Generalization and Details of the WB Algorithm for the Case of Several "Global" Job Classes

In this section the generalization and details of the WB algorithm are given for the case of several "global" job classes, each (possibly) with several "local" jobs, and with four types of servers (as in the BCMP theorem).

For 2 "global" job classes WB showed (using Generating Function approach) an algorithm to

compute the normalization constant C, for server type 2 (P.S.), by using the recursive formulae:

$$G_i(j,k) = G_{i-1}(j,k) + \rho_{i,1}\, G_i(j-1,k) + \rho_{i,2}\, G_i(j,k-1)$$

Where: $C = G_M(L,N)$

M - number of nodes.

L - total number of jobs of class 1.

N - total number of jobs of class 2.

$$\rho_{i,p} = \frac{\lambda_{i,p}}{\mu_{i,p}}$$

p - "global" job class

Unfortunately, in that paper, they didn't show the algorithm to initialize the G(j,k) matrix, how to treat the G(0,k) and G(j,0) cases, and most important - how to generalize for the other 3 types of the BCMP servers and how to handle "local" job classes and more than two "global" job classes. As the solutions for all these probiems is needed to evaluate efficiently the Multi - Cluster Cm* system, we'll investigate these problems in the two following subsections.

III.4.1 Algorithm for Two "Global" Job Classes

The recursive formulae given before was derived using the Generating Function approach. This method will be followed to derive all the necessary initialization elements in the G matrix. The treatment of servers type 1 and 4 is identical to that of server type 2 (This follows immediately from the fact that the general $F_i(s_i)$ equation for all these servers is identical). Treatment of server type 3 is subsequently given. The general form of the G matrix:

No. in each class \ Stage	1	2		i		M
0,0						
1,0						
.						
L,0						
0,1						
.						
L,1						
.						
L,N						G(L,N) = C

The Generating Function for a server (node) of type 1,2,4 (with t,u as the generating functions variables) is given by:

$$f_i(t,u) = \sum_{h=0}^{\infty} (\rho_{i,1}t + \rho_{i,2}u)^h = 1 + (\rho_{i,1}t + \rho_{i,2}u) + (\rho_{i,1}t + \rho_{i,2}u)^2 + \ldots$$

$$= 1/(1 - \rho_{i,1}t - \rho_{i,2}u)$$

The generating function for the network is given by g(t,u), where:

$$g_h(t,u) = \prod_{i=1}^{h} f_i(t,u) \quad \text{and} \quad g(t,u) = g_M(t,u)$$

We will denote by $G_i(j,k)$ the coefficients of $t^j u^k$ in $g_i(t,u)$. It is easy to see that $C = G_M(L,N)$ is the coefficient of $t^L u^N$ in $g_M(t,u)$ and is received by summing all the possible states of the form:

$$\sum_{\substack{\text{over all possible} \\ \text{states } \sum m_i = L \\ \sum n_i = N}} \prod_{i=1}^{M} \frac{(m_i+n_i)!}{m_i! \cdot n_i!} \rho_{i,1}^{m_i} \cdot \rho_{i,2}^{n_i}$$

which is identical to the definition of the normalization constant C.

The recursive algorithm for $G_i(j,k)$ given above, is derived in WB using the algorithm:

$g_1(t,u) = f_1(t,u)$

$g_i(t,u) = g_{i-1}(t,u)\, f_i(t,u) = g_{i-1}(t,u)\frac{1}{1-\rho_{i,1}t-\rho_{i,2}u}$

$g_i(t,u) = g_{i-1}(t,u) - \rho_{i,1}t\, g_i(t,u) - \rho_{i,2}u\, g_i(t,u)$

and therefore the coefficient of $t^j u^k$ is given by:

$G_i(j,k) = G_{i-1}(j,k) + \rho_{i,1}\, G_i(j-1,k) + \rho_{i,2}\, G_i(j,k-1)$

III.4.2 Initialization

Special cases:

(a) $G_1(0,0)$

$G_1(0,0)$ is the coefficient of $t^0 u^0$ in $g_1(t,u) = f_1(t,u) = \sum_{h=1}^{\infty} (\rho_{1,1}t + \rho_{1,2}u)^h$

therefore only h=0 fulfills this condition and $G_1(0,0) = 1$.

(b) $G_1(j,k)$

$G_1(j,k)$ is the coefficient of $t^j u^k$ in $g_1(t,u) =$

$\sum_{h=1}^{\infty}(\rho_{1,1}t + \rho_{1,2}u)^h = 1/(1 - \rho_{1,1}t + \rho_{1,2}u)$

$g_1(t,u) = \rho_{1,1}\, t\, g_1(t,u) + \rho_{1,2}\, u\, g_1(t,u)$

and $G_1(j,k) = \rho_{1,1}\, G_1(j-1,k) + \rho_{1,2}\, G_1(j,k-1)$

Which is received from the general recursive formulae by letting $G_0(j,k)=0$.

(c) $G_i(0,k)$ and $G_i(j,0)$

For $G_i(0,k)$, we observe again that in the general equation:

$g_i(t,u) = g_{i-1}(t,u) + \rho_{i,1}\, t\, g_i(t,u) + \rho_{i,2}\, u\, g_i(t.u)$

we are looking for the coefficients of $t^0 u^k$ in $g_i(t,u)$, This clearly can't be achieved by the

second term (which includes at least t^1 in all it's components). Therefore:

$G_i(0,k) = G_{i-1}(0,k) + \rho_{i,2}\, G_i(0,k-1)$.

It can be similarly shown that:

$G_i(j,0) = G_{i-1}(j,0) + \rho_{i,1}\, G_i(j-1,0)$

i.e. for all $j<0$ ($j = -1$) or $k<0$ ($k = -1$) we have to set $G_i(j,k)=0$ in the algorithm.

We can, therefore, conclude that the general recursive algorithm holds if we initialize:

$G_1(0,0)=1$, $G_0(j,k)=0$ and $G_i(j,k)=0$ for all $j<0$ or $k<0$.

III.4.3 Extension for Several "Global" Job Classes for Servers Types 1,2 and 4

The proof here is derived similarly to that in the previous section. We'll derive, as an example, the algorithm for the case of four "global" job classes with m, n, o, p jobs in each. The correctness of this algorithm for an arbitrary number of "global" jobs follows immediately.

The probability of the aggregate state of number of customers of each job class in each server (m_i,n_i,o_i,p_i), can be shown (from the balance equations) to be:

$$\Pr(m_1,n_1,o_1,p_1;\ m_2,n_2,o_2,p_2;\ \ldots,\ m_M,n_M,o_M,p_M) =$$

$$= (1/C) \prod_{i=1}^{M} \frac{(m_i+n_i+o_i+p_i)!}{m_i!\cdot n_i!\cdot o_i!\cdot p_i!}\ \rho_{i,1}^{m}\ \rho_{i,2}^{n}\ \rho_{i,3}^{o}\ \rho_{i,4}^{p}$$

$$\text{Where } C = \sum_{\substack{\text{over all } m_i,n_i,o_i,p_i,\\ \sum m_i=m,\ \sum n_i=n,\ \sum o_i=o,\\ \sum p_i=p}} \prod_{i=1}^{M} \frac{(m_i+n_i+o_i+p_i)!}{m_i!\cdot n_i!\cdot o_i!\cdot p_i!}\ \rho_{i,1}^{m}\ \rho_{i,2}^{n}\ \rho_{i,3}^{0}\ \rho_{i,4}^{p}$$

and M = number of nodes as before.

The Generating function of the node:

$$f_i(t,u,v,w) = \sum_{h=1}^{\infty} (\rho_{i,1}t + \rho_{i,2}u + \rho_{i,3}v + \rho_{i,4}w)^h =$$

$$= \frac{1}{1 - (\rho_{i,1}t + \rho_{i,2}u + \rho_{i,3}v + \rho_{i,4}w)}$$

Generating function for the network:

$$g_h(t,u,v,w) = \prod_{i=1}^{h} f_i(t,u,v,w).$$

G(m,n,o,p) - The coefficient of $t^m u^n v^o w^p$ is just C, sum of all the possible terms as given above. The algorithm to compute G(m,n,o,p) is derived as follows:

$g_1(t,u,v,w) = f_1(t,u,v,w)$

$g_i(t,u,v,w) = g_{i-1}(t,u,v,w)\, f_i(t,u,v,w)$, and $g_M(t,u,v,w) = g(t,u,v,w)$

$$g_i(t,u,v,w) = g_{i-1}(t,u,v,w) \frac{1}{1 - (\rho_{i,1} t + \rho_{i,2} u + \rho_{i,3} v + \rho_{i,4} w)}$$

$g_i(t,u,v,w) = g_{i-1}(t,u,v,w) + \rho_{i,1}\, t\, g_i(t,u,v,w) + \rho_{i,2}\, u\, g_i(t,u,v,w) + \rho_{i,3}\, v\, g_i(t,u,v,w) + \rho_{i,4}\, w\, g_i(t,u,v,w)$

Therefore the recursive algorithm for $G_i(j,k,l,q)$ is:

$G_i(j,k,l,q) = G_{i-1}(j,k,l,q) + \rho_{i,1}\, G_i(j-1,k,l,q) + \rho_{i,2}\, G_i(j,k-1,l,q) + \rho_{i,3}\, G_i(j,k,l-1,q) + \rho_{i,4}\, G_i(j,k,l,q-1)$

Where $0 \le j \le m$, $0 \le k \le n$, $0 \le l \le o$, $0 \le q \le p$.

Matrix size and operation count:

There are about L multiplications and additions per step (L = number of "global" job classes). It they all visit each of the M noded in the network then the total number of additions and multiplications is o(M L m n o p) (the exact number is M L (m+2) (n+2) (o+2) (p+2)). The matrix size is (m+2) (n+2) (o+2) (p+2).

Initialization:

In a similar way as in the previous section:

(a) $G_1(0,0,0,0) = 1$

(b) $G_1(j,k,l,q) = \rho_{1,1}\, G_1(j-1,k,l,q) + \rho_{1,2}\, G_1(j,k-1,l,q) + \rho_{1,3}\, G_1(j,k,l-1,q) + \rho_{1,4}\, G_1(j,k,l,q-1)$

(c) $G_i(0,k,l,q] = G_{i-1}(0,k,l,q) + \rho_{i,2}\, G_i(0,k-1,l,q) + \rho_{i,3}\, G_i(0,k,l-1,q) + \rho_{i,4}\, G_i(0,k,l,q-1)$

and similarly for $G_i(j,0,l,q)$, $G_i(j,k,0,q)$ and $G_i(j,k,l,0)$.

From the above we can conclude that the general recursive algorithm can be applied if we initialize $G_0(j,k,l,q) = 0$ for all j,k,l,q and $G_i(j,k,l,q) = 0$ for all $j<0$, $k<0$, $l<0$, $q<0$.

III.5 Proof of the Algorithm for Server Type 3 (I.S)

III.5.1 Solution of Two "Global" Job Classes

The form of the solution of $F_i(s_i) = \frac{\rho_{i,1}^{m_i} \cdot \rho_{i,2}^{n_i}}{m_i! \cdot n_i!}$ leads to a server Generating function of:

$$f_i(t,u) = 1 + \frac{\rho_{i,1}t + \rho_{i,2}u}{1!} + \frac{(\rho_{i,1}t + \rho_{i,2}u)^2}{2!} + \ldots = \sum_{h=0}^{\infty} \frac{(\rho_{i,1}t + \rho_{i,2}u)^h}{h!}$$

The Generating function of the network is defined as before:

$$g_i(t,u) = \prod_{h=1}^{i} f_h(t,u), \text{ and } g(t,u) = g_M(t,u)$$

$G_i(j,k)$ is the coefficient of $t^j u^k$ in $g_i(t,u)$. Clearly, the WB solution does not work for this type of server due to the different Generating function.

We notice that:

$$f_i(t,u) = \sum_{h=0}^{\infty} \frac{1}{h!} (\rho_{i,1}t + \rho_{i,2}u)^h =$$

$$= e^{(\rho_{i,1}t + \rho_{i,2}u)} = e^{(\rho_{i,1}t)} \cdot e^{(\rho_{i,2}u)}$$

Therefore, the algorithm can be developed as follows:

$g_1(t,u) = f_1(t,u)$

$g_i(t,u) = g_{i-1}(t,u)\, f_i(t,u) = g_{i-1}(t,u)\, e^{(\rho_{i,1}t)}\, e^{(\rho_{i,1}u)}$

If the first P nodes are chosen to be of type 3 we get:

$$g_p(t,u) = e^{\left(\sum_{h=1}^{P} \rho_{h,1}\right)\cdot t} \cdot e^{\left(\sum_{h=1}^{P} \rho_{h,2}\right)\cdot u} =$$

$$= \sum_{f=1}^{\infty} \left(t \cdot \sum_{h=1}^{P} (\rho_{h,1}) + u \cdot \sum_{h=1}^{P} (\rho_{h,2}) \right)^{f} \cdot \frac{1}{f!}$$

and $G_p(j,k)$ - the coefficient of $t^j u^k$ is therefore given by:

$$G_p(j,k) = \frac{\left(\sum_{h=1}^{P} \rho_{h,1}\right)^{j} \cdot \left(\sum_{h=1}^{P} \rho_{h,2}\right)^{k}}{j! \cdot k!}$$

Which means that we can save a significant amount of computation by summing up the "local" ρ's for each "global" class and by solving directly for $G_p(j,k)$ using the algorithm above - thus skipping the first P-1 iterations in the G matrix calculation; then continue, as before, with the general algorithm for the other M-P nodes.

III.5.2 Solution of Several "Global" Job Classes (For Server Type 3)

In a similar way to the previous subsection, it is easy to show that:

$$f_i(t,u,v,w) = \sum_{h=0}^{\infty} \frac{1}{h!} \left(\rho_{i,1}t + \rho_{i,2}u + \rho_{i,3}v + \rho_{i,4}w\right)^{h}$$

$$= e^{(\rho_{i,1}\cdot t + \rho_{i,2}\cdot u + \rho_{i,3}\cdot v + \rho_{i,4}\cdot w)}$$

and $g_i(t,u,v,w) = g_{i-1}(t,u,v,w)\, f_i(t,u,v,w)$

If the first P nodes are of type 3 we receive (the coefficients of $t^j u^k v^l w^q$) :

$$G_p(j,k,l,q) = \frac{\left(\sum_{h=1}^{P} \rho_{h,1}\right)^{j} \cdot \left(\sum_{h=1}^{P} \rho_{h,2}\right)^{k} \cdot \left(\sum_{h=1}^{P} \rho_{h,3}\right)^{l} \cdot \left(\sum_{h=1}^{P} \rho_{h,4}\right)^{q}}{j!\; k!\; l!\; q!}$$

III.5.3 Algorithm for the Evaluation of Queue Statistics of Server Type 3

For the Multiprocessor queueing network model, we are interested in deriving the queue length distribution, mean queue length, and throughput for a type 3 node (the CPU'S). We know that in the Cm* queuing network model only one "global" job class exist in that node - which makes the algorithm simpler.

To calculate that queue statistics we will add that type 3 node as the last node (M) i.e.:

$$g_M(t,u,v,w) = g_{M-1}(t,u,v,w)\, f_M(t,u,v,w)$$

$$f_M(t,u,v,w) = \sum_{h=0}^{\infty} \frac{(\rho_M \cdot t)^h}{h!} \quad \text{(only one "global" job in this server)}$$

$$\text{Therefore: } G_M(j,k,l,q) = \sum_{h=0}^{j} G_{m-1}(j-h,k,l,q)\, \frac{(\rho_M)^h}{h!}$$

and C can be directly derived as:

$$C = G_M(m,n,o,p) = \sum_{h=0}^{m} G_{M-1}(m-h,n,o,p)\, \frac{(\rho_M)^h}{h!}$$

To calculate the mean queue length in this server:

$$\overline{Q_M} = \sum_{k=0}^{m} k \cdot Pr(n_M = k).$$

To calculate the probability of finding k jobs in the server ($Pr\{n_M=k\}$), we will use the Generating function again. Defining $h_M(t,u,v,w)$:

$$h_M(t,u,v,w) = g_{M-1}(t,u,v,w)\, \frac{(\rho_M \cdot t)^k}{k!}$$

The probability $Pr(n_M=k)$ is the coefficient of t^m in $h_M(t,u,v,w)$ (sum of all terms with t^k in last term divided by the normalization factor) therefore:

$$Pr(n_M=k) = \frac{G_{M-1}(m-k,n) \cdot \rho_M^k}{G_M(m,n) \cdot k!}$$

and $\overline{Q_M} = \sum_{k=1}^{m} k \frac{G_{M-1}(m-k,n) \cdot \rho_M^k}{G_M(m,n) \cdot k!} = \sum_{k=1}^{m} \frac{G_{M-1}(m-k,n)}{G_M(m,n)} \cdot \frac{\rho_M^k}{(k-1)!}$

From which the throughput can be easily calculated.

III.6 Summary: Algorithm for Four Cm* Clusters (Four "Global" Classes).

Here the algorithm which was used to compute the normalization constant C and Q_n is presented, for the case of four "global" job classes of size m,n,o,p, and S + 1 type 3 nodes out of MAXN total nodes. S nodes type 3 are evaluated first, the (S+1) node is evaluated last.

(a) Define a matrix $G_i(m,n,o,p)$; Set $i \leftarrow S$.

(b) Compute $\rho_{S,r} = \sum_{j=1}^{S} \rho_{j,r}$,

$1 < r < 4$ (the "global" job class) and $\rho_{j,r} = 0$ if "global" job class r does not visit node j.

(c) Compute: $G_i(j,k,l,q) = \frac{(\rho_{S,1})^j \cdot (\rho_{S,2})^k \cdot (\rho_{S,3})^l \cdot (\rho_{S,4})^q}{j! \cdot k! \cdot l! \cdot q!}$

(d) While { (i←i+1) < MAXN } do step (e)

(e) $G_i(j,k,l,q) = G_i(j,k,l,q) + \rho_{i,1} G_i(j-1,k,l,q) + \rho_{i,2} G_i(j,k-1,l,q) + \rho_{i,3} G_i(j,k,l-1,q) + \rho_{i,4} G_i(j,k,l,q-1)$

for all $0 \le j \le m$, $0 \le k \le n$, $0 \le l \le o$, $0 \le q \le p$, and $G_i(j,k,l,q)=0$ if ($j<0$ or $k<0$, or $l<0$, or $q<0$).

(f) $C = G_{MAXN}(m,n,o,p) = \sum_{h=0}^{m} G_1(m-h,n,o,p) \frac{(\rho_{MAXN,1})^h}{h!}$

(the last node is type 3 with only job class 1 visiting it).

(g) $Q_m = (1/C) \sum_{h=1}^{m} \frac{G_1(m-h,n,o,p) \ (\rho_{MAXN,1})^h}{(h-1)!}$

IV. Appendix 4: Muntz and Wong Algorithms for Closed Queueing Networks

IV.1 Introduction

In this appendix the algorithm proposed by Muntz and Wong [MUNTZ 74], is presented. The algorithm was modified to include several classes of customers in a server (which is a direct extension to Muntz and Wong results), and was used as the first computational procedure to solve the one Cluster Cm* models presented in chapter 3 section 3.1. The reader is referred to the original paper for details.

IV.2 The Queueing Network Algorithms

The algorithm presented was used to solve the four server queueing model of Para. 3.2.0.1 . The notation used are similar to those given by [MUNTZ 74].

The probability that the system is in state $S = n_1, n_{2,1}, n_{2,2}, n_{2,3}, n_3, n_4$ (where $n_{i,j}$ is the probability of having n customers of class j in node i, and n_i means that only one class of customers visit this node) is given by:

$$P(S) = (1/C)\, g_1(n_1)\, g_2(n_{2,1}\, n_{2,2}\, n_{2,3})\, g_3(n_3)\, g_4 n_4$$

Where $g_i(n_i)$ depends on the server type (for details see [BASKETT 75] or the F_i functions in appendix 3, which are equivalent to the g_i functions). C is a normalization constant.

(a) Generate the $g_1(n_1) \ldots g_4(n_4)$ with $N \geq n_i \geq 0$ for the 4 service stations in the model, where N is the number of customers in the closed queueing network.

- Note that for service station 2 (which has three classes of customers), $n_2 = n_{2,1} + n_{2,2} + n_{2,3}$, and we have to calculate an aggregate state of $g_2(n_2)$ by using:

$$g_2(n_2) = \sum_{i=0}^{n_2} \sum_{j=0}^{n_{2-i}} g_2(i,j,n_2-i-j);\ g_2(0,0,0) = 1;\ 0 \leq n_2 \leq N$$

(b) let $K = n_1 + n_2$, compute:

$$g_2^{*}(K) = \sum g_1(n_1) \cdot g_2(n_2),$$

for all possible n_1, n_2 values $0 \leq n_1 \leq K$; $0 \leq n_2 \leq K$; $n_1 + n_2 = K$.

To compute we simply use:

$$g_2^*(K) = \sum_{n_1=0}^{k} g_1(n_1) g_2(K-n_1); \quad N \geq (n_1+n_2) = K \geq 0$$

sum in service stations 1 and 2.

and $g_2^*(0) = g_1(0)\, g_2(0) = 1$

(c) For i = 3,4 compute

$$g_i^*(K) = \sum_{j=0}^{K} g_{i-1}^*(j) g_i(K-j); \quad 0 \leq K \leq N$$

(d) $C = 1 / g_4^*(N)$

IV.3 Computation of Queue Length Distribution

We are interested in the queue length distribution in server 1 (the CPU's in the models). Making this server the last service center in the g_i^* calculation (i = 4) allows us to calculate the $Q_M(n_M)$ (marginal probability that service center M has n_M customers) from:

$$Q_M(n_M) = C\, g_{M-1}^*(N - n_M)\, g_M(n_M); \quad N \geq n_M \geq 0$$

The throughput of the model can be easily calculated from the average number in server 1 as was done in the previous models.

V. Appendix 5: List of Performance Model Programs

This appendix contains a list of the performance models programs used for the multiprocessor and computer network investigations. The programs are listed in the same order that they are presented in the thesis. They are available on the CMU PDP10/A on account X335LR20.

Program Description

NCMSTR.SAI - Discrete time simulation model (in subsection 3.2.1).

CMSIMU. - M/M/1//N Poisson server model (in subsection 3.2.2).

MD1.APL - M/D/1//N deterministic server model (in subsection 3.2.3).

NET1.APL - One cluster queuing network model (paragraphs 3.3.2.1, 3.3.2.2).

BIGNET.APL - Multi cluster queuing network model, first cluster (subsection 3.3.3).

SECCLS.APL - Multi cluster queuing network model, second cluster (subsection 3.3.3).

IONET.APL - Multi cluster queuing network system with I/O node. Predicting system performance (subsection 3.4.5).

IONET1.APL - Multi cluster queuing network system with I/O node. Predicting I/O rate (subsection 3.4.5).

IONET5.APL - Multi cluster queuing network system with several I/O block moves in parallel (subsection 3.4.5).

NGRADA.FTN, CONTEN.FTN - Performance vs. memory size trade off (subsection 3.4.6).

NETWRK.APL - Local network queuing model (section 4.5).

References

1. [Baer 73] Baer J.L., "A Survey of Some Theoretical Aspects of Multiprocessors," Computing Surveys, Vol. 5, No. 1, March 1973.

2. [Balas 76] Balas E. and M. Padberg, "Set Partitioning - A Survey," SIAM Review, Vol. 18, No. 4, Oct. 1976.

3. [Baskett 73] Baskett F.S., K.M. Chandy, R.R.Muntz and F.G. Palacios "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," JACM, Vol. 22, No. 2, April 1975, PP 248-260

4. [Baskett 76] Baskett F.S. and A.J. Smith, "Interference in Multiprocessor Computer Systems with Interleved Memory," CACM, Vol. 19, No. 6, June 1976.

5. [Baudet 77] Baudet G.M., "Asynchronous Iterative Methods for Multiprocessors," JACM, Vol. 25, No. 2, April 1978, pp. 226-244.

6. [Baudet 78] Baudet G.M., "The Design and Analysis of Algorithms for Asynchronous Multiprocessors," Department of Computer Science, CMU, TR CMU-CS-78-116, April 1978.

7. [Bhandarkar 1973] Bhandarkar D.P., "Analytical Models for Memory Interference in Multiprocessor Computer System," Ph.D. Dissertation, Dept. of Electrical Engineering, CMU, Sept. 1973.

8. [Bose 1976] Bose C.A. "Validation of A Queueing Network Model with Classes of Customers," Proc. of the Intl. Symp. on Comp. Performance Modelling, Measurement and Evaluation, Harvard Univ., March 1976.

9. [Brantley 77] Brantley W.C., G.W. Leive and D.P. Siewiorek, "Decomposition of Data Flow Graphs on Multiprocessors," NCC Proceedings, Afips Press, Vol. 46, PP. 281-290, 1977.

10. [Brown 76] Brown K.Q., "Simulation of A Cm* Cluster," CMU, Computer Science Department internal memo. May, 1976.

11. [Buzen 73] Buzen J., "Computational Algorithms for Closed Queueing Networks with Exponential Servers," CACM 16, 1973, PP 527-531.

12. [Chandy 77] Chandy K.M., J.H. Howard and D.F. Towsley, "Product Form and Local Balance in Queueing Networks," JACM, Vol. 24, No. 2, April 1977, PP 250-263.

13. [Chansler 77] Chansler R. and P.S. Sindhu "Report on the Image Understanding Problem on Cm*," CMU Computer Science Dept. Technical Report, October 1977.

14. [Eckhouse 78] Eckhouse R.H., J.A. Stankovic and A. Van Dam., "Issues in Distributed Processing - An Overview of Two Workshops," Computer, January 1978.

15. [Enslow 77a] Enslow P.H. "Multiprocessor Organization - A Survey," Computing Surveys, Vol. 9, No. 1, March 1977.

16. [Enslow 77b] Enslow P.H., "Distributed Data Processing Systems: Some Definitions and Research Issues," Workshop of Distributed Processing, Brown University, August 1977.

17. [Enslow 78] Enslow P.H., "What is a Distributed Data Processing System," Computer, January 1978.

18. [Farber 75] Farber D.J. "A Ring Network," Datamation, Feb. 1975.

19. [Feiler 77] Feiler P., "Harpy, Speech Recognition System," in [Fuller 77b].

20. [Fishman 73] Fishman G., "Discrete Digital Simulation," John Wiley & Sons inc., 1973.

21. [Flynn 72] Flynn M.J., "Some Computer Organizations and Their Effectiveness," IEEE Trans. on Comp., Vol. C-21, No. 9, Sept. 1972.

22. [Fraser 75] Fraser D.J., "A Virtual Channel Network," Datamation, Feb. 1975.

23. [Fuller 77a] Fuller S.H. et al, "A Collection of Papers on Cm*, A Multi-Microprocessor Computer System," Comp. Science Dept., CMU, Feb. 1977.

24. [Fuller 77b] Fuller S.H., A.K Jones and I. Durham (editors), "Cm* Review, June 1977," CMU Comp. Science Dept. Technical Report, June 1977.

25. [Fuller 78] Fuller S.H., J.K. Ousterhout, L. Raskin, P.I. Rubinfeld, P.S. Sindhu, R.J. Swan, "Multi-Microprocessors: an Overview and Working Example," Proceeding of the IEEE, Feb. 1978.

26. [Gelenbe 76] Gelenbe E. and R.R. Muntz, "Probabalistics Models of Computer Systems - Part 1 (Exact Results)," Acta Informatica 7, Springer-Verlag 1976, pp 35 - 60.

27. [Giammo 76] Giammo T., "Validation of A Computer performance Model of the Queueing Network Family," Proc. of the Intl. Symp. on Comp. Performance, Modelling and Evaluation, Harvard Univ., March 1976.

28. [Gonzales 78] Gonzales M.J., "Future Directions in Computer Architecture," Computer, March 1978.

29. [Gordon 67] Gordon W.J. and G.F. Newell, "Closed Queueing Systems with Exponential Servers," Operation Research 15, 1967, PP 254-265.

30. [Hibbard 78] Hibbard P., A. Hisgen and T. Rodenheffer "A Language Implementation Design for a Multiprocessor Computer System," 5th Comp. Arch. Conf., April 1978.

31. [Hoogendoorn 77] Hoogendoorn C.H., Reduction of Memory Interference in Multiprocessor Systems," Conf. Proc. of the 4th. Annual Symposium on Comp. Architecture, March 1977.

32. [Jackson 63] Jackson J.R., "Jobshop - like Queueing System," Management Science

10, 1963 , PP 131-142.

33. [Jaiswal 68] Jaiswal N.K., "Priority Queues," Academic Press, New York, 1968.

34. [Jensen 77] Jensen E.D. and G.A. Anderson, "Computer Interconnection Structures: Taxonomy, Characteristics and Examples," Computing Surveys, Vol. 7, No. 4, PP. 197-213, Dec. 1977.

35. [Jensen 78] Jensen E.D., "The Honeywell Experimental Distributed Processor - an Overview," Computer, January 1978.

36. [Jones 78] Jones A.K., R.J. Chansler, I. Durham, P.H. Feiler, D.A. Scelza, K. Schwans and S.R. Vegdahl, "Programming Issues Raised by a Multiprocessor," Proceeding of the IEEE, Vol. 66, No.2, February 1978, pp 229-237.

37. [Kimbelton 75] Kimbelton S.R. and G.M. Schneider, "Computer Communications Networks, Approaches, Objectives, and Performance Considerations," Computing Surveys, Vol. 7, No. 3, Sept. 1975.

38. [Kiney 78] Kiney L.L. and R.G. Arnold, "Analysis of a Multiprocessor System with a Shared Bus," Proceeding of the 5th. Conf. on Computer Architecture, April 1978.

39. [Kleinrock 75] Kleinrock L., "Queueing systems, Volume 1: Theory," John Wiley & Sons inc., 1975.

40. [Kleinrock 76] Kleinrock L., "Queueing Systems, Volume 2: Computer Applications, " John Wiley & Sons inc., 1976.

41. [Kung 76] Kung H.T., "Synchronized and Asynchronous Parallel Algorithms for Multiprocessors", in Algorithms and Complexity, New Directions and Recent Results, ed. by J.F. Traub, Academic Press, New York, 1976, pp. 153-200.

42. [Levy 73] Levy J.V., "Computing with Multiple Processors," SLAC Report 161, Stanford Univ., April 1973.

43. [Lipsky 1977] Lipsky L. and J.D. Church, "Applications of A Queueing Network Model for a Computer System," Computing Surveys, Vol. 9, No. 3, Sept. 1977.

44. [Liu 75] Liu T.M. and C.C. Reames, "The Design of the Distributed Loop Computer Network," Proceeding of the Intl. Comp. Symposium, 1975 (Vol. 1).

45. [Liu 77] Liu T. M., R. Pardo and G. Babic, "A performance Study of Distributed Control Loop Networks," Proceeding of the 1977 Intl. Conf. of Parallel Processing, Bellaire, Michigan, August 77.

46. [Lowerre 76] Lowerre B.T., "The Harpy Speech Recognition System" CMU, Computer Science Department TR, April 1976.

47. [Marathe 77] Marathe M. "Performance Evaluation at the Hardware Level and the Operating System Kernel Design Level", Ph.D. Dissertation, Comp. Science Dept., CMU, Dec. 1977.

48. [Mazare 77] Mazare G., "A Few Examples of How to Use A Symmetrical Multi-Microprocessor," Conf. Proc. of the 4th. Annual Symposium on Comp. Architecture, March 1977.

49. [Metcalfe 76] Metcalfe R.M. and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," CACM, Vol. 19, No. 7, July 1976.

50. [Mudge 77] Mudge J.C., "Design Decisions Acheive Price/Performance Balance in Mid-Range Minicomputers," Computer Design, August 1977.

51. [Muntz 74] Muntz R.R. and J.W. Wong, "Efficient Computational Procedures for Closed Queueing Network Models," Proceedings of the 7th. Hawaii Conf. on System Science, 1974, PP 33-36.

52. [Oleinick 78] Oleinick P., "The implementation of Parallel algorithms on an asynchronous multiprocessor", Ph.D. Dissertation, CMU, Electrical Engineering Dept., July 1978.

53. [Parzen 60] Parzen E., "Modern Probability Theory," John Wiley & Sons inc., 1960.

54. [Peterson 77] "Petri Nets," Computing Surveys, Vol. 9, No. 3, September 1977.

55. [Price 1976] Price T.G., "A Note on the Effect of the Central Processor Service Time Distribution on Processor Utilization in Multiprogrammed Computer Systems," JACM, Vol. 23, No. 2, April 1976, PP 342-346.

56. [Raskin 77] Raskin L., "Performance of a Stand Alone Cm* System," in [Fuller 77b].

57. [Reiser 73] Reiser M. and H. Kobayashi, "Recursive Algorithms for General Queueing Networks with Exponential Servers," IBM Thomas J. Watson Research Center report RC 4254, March 1973.

58. [Reiser 75] Reiser M. and H. Kobayashi, "Queuing Networks with Multiple Closed Chains: Theory and Computational Algorithms," IBM J. Res. Develop. , May 1975.

59. [Reiser 76] Reiser M. "Numerical Methods in Separable Queueing Networks," IBM Thomas J. Watson Research Center Research Report 5842, May 1976.

60. [Sauer 76] Sauer C.H., M. Reiser and E.A. MacNair, "RESQ - A Package for Solution of Generalized Queueing Networks," IBM Thomas J. Watson Research Center Report RC 6462, Nov. 1976.

61. [Sauer 77] Sauer C.H. and E.A. MacNair, "Computer/Communication System Modelling with Extended Queueing Networks," IBM Thomas J. Watson Research Center Report RC 6654, July 1977.

62. [Sherman 77] Sherman E., "Implementation Distributed Networks with SDLC," Digital Design, March 1977.

63. [Singleton 69] Singleton R.C., "Algorithm 347," CACM, Vol. 13., No. 3, March 1969.

64. [Stone 71] Stone H.S., "Parallel Processing with the Perfect Shuffle," IEEE Trans. on Comp., Vol. C-20, No. 2, Feb. 1971.

65. [Strecker 76] Strecker W.D., "Cache Memories for PDP 11 Family computers," Proceedings of the 3rd Annual Symposium on Computer Architecture, 1976, PP 155-158.

66. [Swan 77a] Swan R.J., S.H. Fuller and D.P. Siewiorek, "Cm*: a Modular Multi-Microprocessor," NCC Proceedings, Dallas, Texas, 1977.

67. [Swan 77b] Swan R.J., A. Bechtolsheim, K. Lai, J.K. Ousterhout, "The Implementation of the Cm* Multi-Microprocessor," NCC Proceedings, Dallas, Texas, 1977.

68. [Swan 78] Swan R.J., "The Switching Structure and Addressing Architecture of an Extensible Multiprocessor: Cm*," Ph.D. Dissertation, Comp. Science Dept., CMU, May 1978.

69. [Svobodova 76] Svobodova L., "Computer Performance Measurement and Evaluation Methods: Analysis and Applications," Computer Design and Architecture Series, American Elsevier Publishing C., 1976.

70. [Williams 76] Williams A.C. and R.A. Bhandiwad, "A Generating Function Approach to Queueing Network Analysis of Multiprogrammed Computers," Network, 6 (John Wiley & Sons, Inc.), 1976, P 1-22.

71. [Wulf 72] Wulf W.A. and C.G.Bell, "C.mmp A Multi-Mini-Processor," AFIPS Conf. Proc., Vol. 41, Part 2, FJCC 1972.

72. [Wulf 74] Wulf W.A., Cohen E., Corwin W., Jones A., Levin R., Pierson C., and Pollack F., "Hydra: the Kernel of a Multiprocessor Operating System," CACM, Vol. 17, No. 6, June 1974, pp 337-345.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-78-141	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PERFORMANCE EVALUATION OF MULTIPLE PROCESSOR SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Levy Raskin		8. CONTRACT OR GRANT NUMBER(s) N00014-77-C-0500
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Electrical Engineering Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE August 1978 13. NUMBER OF PAGES 237
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same as above		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Technological trends in semiconductor and micro-processor development are clearly leading towards the production of a "computer on a chip". In the near future such a *computer module* will include the equivalent of today's mini-computer with some memory. Connecting many computer-modules together will probably be a cost-effective way to build high performance computer systems. This thesis investigates and contrasts the performance of

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

two multiple computer-module structures: a multiprocessor with shared memory and a local computer network in which all communication is via messages. Results are derived both from running benchmark applications and from performance models.

Very little knowledge and experience has been gained on the performance of actual multiple processor applications. To investigate the problems and potential of these computer-module structures, an multiple processor system -- Cm* -- has been built at Carnegie-Mellon University. Firmware changes enable the emulation of both an efficient multiprocessor and a local computer network. Experiments were conducted on both types of structures to obtain performance information. Our practical methodology includes measurement of performance parameters using a set of benchmark application programs executing on Cm*, and performance models that were derived and validated - using the measurement results - and then applied for the performance investigation.